This chapter describes those data structures and utility functions that are used throughout the Apple Open Collaborative Environment (AOCE) but are not specific to any one particular manager or package. It describes the data structures you need to be familiar with to use the AOCE toolbox functions and shows you how to use the AOCE utility functions to manipulate these data structures in various ways.

You should read this chapter if you will be using the Standard Mail or Standard Catalog Packages to add AOCE services to your application, are developing a stand-alone mail or communications package that will use AOCE services, or if you are developing lower-level AOCE entities such as catalog services access modules.

Before reading this chapter you should have at least a general understanding of the Apple Open Collaborative Environment. At the minimum, you should have read the chapter "Introduction to AOCE," earlier in this book, which explains the organization and use of the various AOCE managers and services.

About the AOCE Utilities

The AOCE toolbox contains over 60 utility functions that are designed to provide you with easy methods for performing various tasks using the AOCE data structures. Here are some of the services that the AOCE utility functions provide:

- converting data structures to their packed forms (packing)
- converting data structures from their packed forms (unpacking)
- checking data structures to verify that they are in the proper format and contain valid data for their particular type
- comparing data structures for equality
- copying the contents of one data structure to another
- converting variables from one data type to another
- determining the size of data structures
- determining whether a given data structure is null or empty

Unless otherwise noted, all of the AOCE utility functions described in this chapter can be called at interrupt level and do not allocate any memory.

AOCE Data Structures of Maximum and Minimum Size

Some of the AOCE data structures are defined as maximum- or minimum-sized structures. A maximum-sized structure is one that, upon creation, contains enough storage to hold the maximum amount of data possible for that particular type of data structure. An example of a maximum-sized AOCE structure is the `RString` structure shown here and defined on page 2-20.

```
struct RString
{
   RStringHeader
   Byte  body[kRStringMaxBytes];
};
```

When you create a new `RString` structure, it contains enough memory to hold 256 bytes of data in its `body` field, plus the number of bytes necessary for the `RStringHeader` field. You never need to allocate any additional memory for the structure.

By contrast, a minimum-sized structure is one that, upon creation, contains only the minimum necessary storage. The minimum storage varies according to the type of data structure. An example of a minimum-sized structure is the `ProtoRString` structure shown here and defined on page 2-22.

```
struct ProtoRString
{
   RStringHeader
};
```

As you can see, the `ProtoRString` structure differs from the `RString` structure in that it does not contain a `body` field. Therefore, when you create a `ProtoRString` structure for the first time, it contains only enough memory to hold the information in its `RStringHeader` field. If you want to store any additional data in the `ProtoRString` structure, you will have to allocate the memory. See the section "Allocating AOCE Strings of Nonstandard Sizes" on page 2-16 for details on how to allocate additional memory for a `ProtoRString` structure.

The advantage of using minimum-sized AOCE data structures is that you can allocate structures of any size and can save memory by allocating structures that are exactly the size you need. The disadvantage of using minimum-sized AOCE data structures is that you will have to remember to allocate additional storage for the structure as you need it, and you will have to write more code to allocate each structure.

After declaring a variable as a minimum-sized AOCE structure, you may sometimes find that you need to allocate it as a maximum-sized structure. See the section "Allocating a RecordID Structure of Maximum Size" on page 2-16 for more information.

Using the AOCE Utilities

This section describes how you can use various AOCE utility functions and data structures in your own code. Many of the AOCE utility functions have similar characteristics and can be grouped according to the type of operations they perform. This section explains most of the major groups of AOCE utility functions and provides you with background knowledge that may help you understand how to use these functions.

Determining Whether the Collaboration Toolbox Is Available

Before calling any of the AOCE Utility functions, you should verify that the Collaboration toolbox is available by calling the `Gestalt` function with the selector `gestaltOCEToolboxAttr`. If the Collaboration toolbox is present but not running (for example, if the user deactivated it from the PowerTalk Setup control panel), the `Gestalt` function sets the bit `gestaltOCETBPresent` in the `response` parameter. If the Collaboration toolbox is running and available, the function sets the bit `gestaltOCETBAvailable` in the `response` parameter. The Gestalt Manager is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*.

Packing and Unpacking the AOCE Data Structures

Several of the AOCE data structures contain fields that are themselves structures, and these may in turn contain other nested structures. It is sometimes useful to compact, or "flatten" a complex data structure into a sequence of bytes in order to perform an operation more efficiently. This process is known as ***packing*** the data structure. Similarly, the process of reconstructing a data structure from a sequence of bytes is known as ***unpacking*** the data structure.

Many of the AOCE functions pass packed structures. Because the packed forms of these structures are private, you can't read or write them unless you use the utility routines to pack and unpack them.

Another reason for using the packed form of a data structure is to simplify I/O related tasks, such as writing the information contained in a data structure to a file, or sending the data to a serial port. In its packed form, the data is usually just a stream of bytes, which is much easier to work with in I/O operations.

The AOCE toolbox simplifies the processes of packing and unpacking by providing unpacked and packed forms of many of its data structures, as well as the utility functions to convert between the two forms. All of the AOCE packing functions begin with the letters `OCEPack` followed by the name of the data structure they pack, and all of the AOCE unpacking functions begin with the letters `OCEUnpack` followed by the name of the data structure they unpack. For example, the AOCE packing function that packs `RecordID` structures is named `OCEPackRecordID`.

Table 2-1 shows the AOCE data structures that have packed forms, along with the functions used to convert between the packed and unpacked forms.

Table 2-1 AOCE packed data structures and functions used to pack and unpack them

Unpacked data structure	Packed data structure	Packing/unpacking functions
`RString`	`PackedPathName`	`OCEUnpackPathName`
		`OCEPackPathName`
`RLI`	`PackedRLI`	`OCEPackRLI`
		`OCEUnpackRLI`
		`OCEPackedRLIPartsSize`
		`OCEPackRLIParts`
`RecordID`	`PackedRecordID`	`OCEPackRecordID`
		`OCEUnpackRecordID`
`DSSpec`	`PackedDSSpec`	`OCEPackDSSpec`
		`OCEUnpackDSSpec`

Note
The `Rstring` structure is shown in Table 2-1 as the unpacked form of the `PackedPathName` structure. This is actually a special case because the unpacked form of the `PackedPathName` structure is an array of `RString` structures. See the description of the `PackedPathName` structure on page 2-29 for more information. To create a `PackedPathName` structure, you need to supply an array of `RString` structures to the `OCEPackPathName` function (page 2-60). ◆

Unpacking Catalog Specifications

The catalog services specification data structure, of data type `DSSpec`, is central to accessing information within PowerTalk. Unpacking a `PackedDSSpec` structure is the process of converting the sequence of bytes in a `PackedDSSpec` structure into the structure of a `DSSpec`. In its packed form, the `DSSpec` structure contains other data structures that are also packed, so you must unpack each component as well as the `PackedDSSpec` structure itself.

Listing 2-1 shows how to unpack a `DSSpec` structure completely into its component parts, including its nested packed structures.

1. First, allocate a `DSSpec` structure (`DSSpecDumpRecord`) in which to store the contents of the `PackedDSSpec` structure when you unpack it. Also declare Boolean variables to record whether the various parts of the structure are valid.

2. Call the Standard Catalog Package function `SDPGetPanelSelectionSize` to obtain the size of the `PackedDSSpec` structure and then allocate memory for it. Call `SDPGetPanelSelection` to retrieve the `PackedDSSpec` structure.
3. Call the `UnpackPackedDSSpec` function to unpack the `PackedDSSpec` structure. Pass the function a pointer to the `PackedDSSpec` structure to be unpacked and a pointer to the `DSSpec` structure to hold its component parts.
4. Call the `DoDisplayDSSpecDumpRecord` function (which is not shown here) to use the information that you have retrieved from the `PackedDSSpec` structure; for example, to display the contents of a record that a user has selected.
5. It is possible that the `PackedDSSpec` structure you obtained from the `SDPGetPanelSelection` routine contains corrupted data. Therefore, you should check the integrity of the `PackedDSSpec` structure and of each of the nested packed structures that it contains before unpacking them. The `DoUnpackPackedDSSpec` function calls a series of AOCE utility functions to verify the integrity of the packed structures and to unpack them. The validation functions are nested in conditional statements. If any of the structures is invalid, the code prints an error message specifying which structure was corrupted. (The error messages are in the `else` statements at the end of Listing 2-1.)
   - The `OCEValidPackedDSSpec` and `OCEUnpackDSSpec` functions verify and unpack the packed `DSSpec` structure itself. The `OCEGetDSSpecInfo` function returns the type of the `DSSpec` structure.
   - The `OCEValidPackedRLI` and `OCEUnpackRLI` functions verify and unpack the `packedRLI` structure contained in the unpacked `DSSpec` structure.
   - The unpacked `RLI` structure contains a `PackedPathName` structure that you must unpack. However, before unpacking it, you call `OCEDNodeNameCount` to obtain the presumed number of pathnames. Then you allocate a vector to hold the `RString` structures that make up the pathname list. Finally, you call `OCEUnpackPathName` to unpack the `PackedPathName` buffer. If the presumed number of pathnames matches the actual number returned by `OCEUnpackPathName`, you are done.

Listing 2-1 Unpacking a DSSpec structure

```
/* In the example, the following external functions are defined:
   DoNOTE(message)           Write the message to the error log.
   DoFailOSErr(status, msg)  If status is not noErr, begin error recovery.
   DoFailNIL(ptr)            If ptr is nil, begin error recovery. This is
                            generally an unexpected, serious, error.

 The argument PackedDSSpec is stored in a private structure,
 DSSpecDumpRecord. Members of this structure contain pointers to the
 packed DSSpec.*/

typedef struct DSSpecDumpRecord {
  DSSpec          theDSSpec;         /* Unpacked DSSpec */
```

```
  RecordID        recordID;          /* Its record ID structure */
  RLI             theDSSpecRLI;      /* Its unpacked Record Location Info */
  OSType          specType;          /* The type of this DSSpec */
  unsigned short  nodeNameCount;     /* Presumed number of pathnames */
  unsigned short  trueNodeNameCount; /* Actual number of pathnames */
  RStringPtr      *partsVector;      /* -> vector of pathname RStrings */

/* These Boolean variables record the status of the DSSpec. They are true if
   the associated part of the structure is present and in good condition. */

  Boolean     isValidDSSpec;         /* OCEValidPackedDSSpec succeeds */
  Boolean     isNonNullRLI;          /* RLI is present in this DSSpec */
  Boolean     isValidPackedRLI;      /* OCEValidPackedRLI succeeds */
  Boolean     isValidPackedPathName; /* OCEValidPackedPathName succeeds */
  Boolean     isValidUnpackedCount;  /* Unpacked count == presumed count */
} DSSpecDumpRecord, *DSSpecDumpPtr;

void
DoUnpackSDPPanelSelection(
        register DocumentPtr            dbp,
        SDPPanelHandle                  thePanel
    )
{
        OSErr                           status;
        PackedDSSpecPtr                 packedDSSpec;
        unsigned short                  packedDSSpecSize;
        DSSpecDumpRecord                dumpRecord;


/* Allocate memory for the DSSpec and get it from the Standard
   Directory Manager. */

        status = SDPGetPanelSelectionSize(thePanel, &DpackedDSSpecSize);
        DoFailOSErr(status, "\pSDPGetPanelSelectionSize");
        packedDSSpec = (PackedDSSpecPtr) NewPtrClear(packedDSSpecSize);
        DoFailNIL(packedDSSpec );
        status = SDPGetPanelSelection(thePanel, packedDSSpec);
        DoFailOSErr(status, "\pSDPGetPanelSelection");
        DoUnpackPackedDSSpec(packedDSSpec, &dumpRecord);
        DoDisplayDSSpecDumpRecord(&dumpRecord); /* Not shown */
        if (dumpRecord.partsVector != NULL)
            DisposePtr((Ptr) dumpRecord.partsVector);
        if (packedDSSpec != NULL)
```

```
            DisposePtr((Ptr) packedDSSpec);
}

void
DoUnpackPackedDSSpec(
        PackedDSSpecPtr                 packedDSSpec
        register DSSpecDumpPtr          theDSSpecDumpPtr
    )
{
#define SPEC (*theDSSpecDumpPtr)

        ClearMemory(&SPEC, sizeof SPEC);
        SPEC.isValidDSSpec = OCEValidPackedDSSpec(packedDSSpec);
        if (SPEC.isValidDSSpec) {
            OCEUnpackDSSpec(packedDSSpec, &SPEC.theDSSpec, &SPEC.recordID);
            SPEC.specType = OCEGetDSSpecInfo(&SPEC.theDSSpec);
            SPEC.isNonNullRLI = (SPEC.recordID.rli != NULL);
            if (SPEC.isNonNullRLI) {
                SPEC.isValidPackedRLI = OCEValidPackedRLI(SPEC.recordID.rli);
                if (SPEC.isValidPackedRLI) {
                    OCEUnpackRLI(SPEC.recordID.rli, &SPEC.theDSSpecRLI);
                    SPEC.isValidPackedPathName =
                        OCEValidPackedPathName(SPEC.theDSSpecRLI.path);

                     /* SPEC.isValidPackedPathName is false if you click
                        on a printer or CPU in the AppleTalk directory. */

                    if (SPEC.isValidPackedPathName) {
                        SPEC.nodeNameCount =
                           OCEDNodeNameCount(SPEC.theDSSpecRLI.path);

                         /* Allocate a vector to hold the RStrings that make
                            up the pathname list. Then unpack the pathname
                            list. */
                        SPEC.partsPtr = (RStringPtr *) NewPtrClear(
                                sizeof (RStringPtr) * SPEC.nodeNameCount
                            );
                        DoFailNIL(SPEC.partsPtr);
                        SPEC.trueNodeNameCount= OCEUnpackPathName(
                                SPEC.theDSSpecRLI.path,
                                SPEC.partsPtr,
                                SPEC.nodeNameCount
                            );
```

```
                if (SPEC.nodeNameCount == SPEC.trueNodeNameCount)
                    SPEC.isValidUnpackedCount = true;
                else {
                    NOTE("\pUnpacked Node Name Count != Node Name
                          Count");
                }
                else {
                    NOTE("\pInvalid PackedPathName");
                }
            }
            else {
                NOTE("\pInvalid Packed RLI");
            }
        }
        else {
            NOTE("\pValid DSSpec but NULL RLI");
        }
    }
    else {
        NOTE("\pInvalid Packed DSSpec");
    }
}
```

Validating the AOCE Data Structures

The AOCE toolbox provides a set of validation functions that allow you to verify the integrity of the various AOCE data structures. All of the AOCE validation functions begin with the letters "`OCEValid`" and are followed by the name of the data structure that they validate. For example, the AOCE validation function for `PackedDSSpec` structures is called `OCEValidPackedDSSpec`. Table 2-1 on page 2-6 shows the AOCE validation functions along with the data structures that each function validates. You should use the AOCE validation functions whenever you want to make sure that the AOCE data structures allocated in your program

- are valid values for that data type
- contain fields that have valid values
- are of a valid size
- contain fields of a valid size

The way the AOCE validation functions verify the integrity of a data structure depends upon the type of structure being examined. In general, however, AOCE validation functions perform the following checks:

- They determine whether the pointer to the data structure is `nil` or the data structure has a length of 0 and whether these are permissible values for this data structure.

- They determine if the data structure or any of its fields contain values that are not valid for that particular data structure.
- They determine if the value contained in any length fields of the data structure is equal to the number of bytes of data actually contained in that field.
- If the data structure contains fields that are other AOCE data structures, then the validation function passes these fields to other AOCE validation functions until all of the data structure's fields are checked. If the AOCE validation function cannot validate a field, it does not check that field but does check the rest of the data structure for validity.
- For packed data structures, the AOCE validation functions check that the packed data structure is at least as large or larger than the smallest possible packed structure of that type. This ensures that the data structure is at least large enough to hold the minimum amount of data in all of its fields.

Table 2-2 AOCE validation functions and associated data structures

Verify function name	Data structure verified
`OCEValidRString`	`RString`
`OCEValidPackedPathName`	`PackedPathName`
`OCEValidRLI`	`RLI`
`OCEValidPackedRLI`	`PackedRLI`
`OCEValidPackedRecordID`	`PackedRecordID`
`OCEValidPackedDSSpec`	`PackedDSSpec`

Listing 2-2 shows how to use the `OCEValidPackedPathName` function (page 2-62) to compare a `PackedPathName` structure for validity. This sample code calls the `OCEValidPackedPathName` function two different times to illustrate cases when the `PackedPathName` structure is valid and when it is not valid. The `MyValidatePackedPathName` function assumes the existence of a routine named `DoErrorChecking`, which handles any memory errors. For information on the `PackedPathName` structure see page 2-29.

Listing 2-2 Validating a `PackedPathName` structure

```
MyValidatePackedPathName()
{
   PackedPathName*      myPackedPathName;
   PackedPathName*      myNilPackedPathName;
   Boolean              isValid;       /* value returned by
                                          OCEValidPackedPathName/*
```

```
   /* First call OCEValidPackedPathName with a nil pointer. */
   myNilPackedPathName = nil;

   /* The AOCE toolbox does not consider nil PackedPathName
      pointers to be valid, so this call to OCEValidPackedPathName
      returns false in the isValid variable. */

   isValid = OCEValidPackedPathName(myNilPackedPathName);

   /* Allocate a PackedPathName structure. */
   myPackedPathName = (PackedPathName *)
                        NewPtr(sizeof(PackedPathName);

   DoErrorChecking();   /* make sure the PackedPathName allocation
                           didn't fail */

   myPackedPathName->dataLength = 0;/* set the length of the
                                       PackedPathName to 0 */

   /* The AOCE toolbox considers a PackedPathName with a length of
      0 to be valid, so this call to OCEValidPackedPathName
      returns true in the isValid variable. */

   isValid = OCEValidPackedPathName(myPackedPathName);
}
```

Comparing AOCE Data Structures for Equality

The AOCE toolbox provides a set of functions that allow you to compare the AOCE data structures for equality. All the AOCE equality functions begin with the letters `OCEEqual` and are followed by the type of the data structures being compared. For example, the AOCE equality function that compares two `RString` structures is called `OCEEqualRString`. The AOCE equality functions and the data structures that they compare are shown in Table 2-1 on page 2-6.

The actual method used to determine the equality of the data structures varies with their type. Before using any equality function, you should read its description to find out exactly how that function compares the data structures for equality. For example, the `OCEEqualPackedPathName` function (page 2-61) considers two `PackedPathName` structures to be equal if these three conditions are met: (a) one of the pointers passed into the function is `nil`, (b) the other pointer is not `nil`, and (c) the pointer that is not `nil` does point to a `PackedPathName` structure that has a length of 0. In general, each AOCE equality function acts as follows when comparing two structures for equality:

- If the data structures are packed, then the AOCE equality function unpacks them before comparing them. This has no effect on the original data structures.
- If the pointers to the data structures are both `nil`, then they are equal.

- If the data structures are not the same length, then they are not equal and no further comparisons are performed on them.
- If the data structures have fields that are other AOCE data structures, then the AOCE equality function compares these nested structures by calling the appropriate AOCE equality functions for these data structure types. This process is repeated for each nested data structure. If any of the nested structures are not equal, then the AOCE equality function returns `false`, indicating that the original data structures are not equal.

Table 2-3 AOCE equality functions and associated data structures

Equality Function Name	Data Structures Compared
`OCEEqualRString`	`RString`
`OCEEqualCreationID`	`CreationID`
`OCEEqualPackedPathName`	`PackedPathName`
`OCEEqualDirDiscriminator`	`DirDiscriminator`
`OCEEqualRLI`	`RLI`
`OCEEqualPackedRLI`	`PackedRLI`
`OCEEqualLocalRecordID`	`LocalRecordID`
`OCEEqualShortRecordID`	`ShortRecordID`
`OCEEqualRecordID`	`RecordID`
`OCEEqualPackedRecordID`	`PackedRecordID`
`OCEEqualDSSpec`	`DSSpec`
`OCEEqualPackedDSSpec`	`PackedDSSpec`

Copying AOCE Data Structures

The AOCE toolbox provides a set of functions for copying the contents of one AOCE data structure into another. You should use the AOCE copy functions whenever you want to copy the contents of one AOCE data structure into another.

None of the utility functions allocates any memory. Therefore, before you call an AOCE copy function, you need to make sure you have allocated both the source and destination structures. The AOCE copy function returns an error if the structures you allocate are too small. You should always check the value returned by an AOCE copy function to make sure that the copy took place successfully.

All of the AOCE copy functions begin with the letters `OCECopy` and are followed by the name of the data structure type that they copy. For example, the AOCE function for copying two `CreationID` structures is `OCECopyCreationID`. See Table 2-1 on page 2-6 for a list of the AOCE copy functions and the data structures that they copy.

Listing 2-3 illustrates the correct way to call an AOCE copy function. The `MyCopyingCode` function uses the `OCECopyRString` (page 2-45) utility routine to copy the `sourceRString` structure. The `sourceRString` structure is assumed to be a valid `RString` structure that has already been allocated and initialized elsewhere. The `MyCopyingCode` function also uses the Macintosh toolbox routine `MemErr` to check for memory allocation errors. In addition, the `myCopyingCode` function assumes the existence of a function named `DoErrorHandling` that handles an error if one occurs.

Listing 2-3 Calling a copy function

```
MyCopyingCode(RString*  sourceRString)
{
   /* This function assumes that the sourceRString parameter is
      a pointer to a valid RString containing data to be copied.
*/

   OSErr    myError;    /* this variable holds the value returned
                           by the OCECopyRString function */


   RString* destinationRString;  /* pointer to the RString that
                                    you want to copy the contents
                                    of sourceRString into */
   destinationRString = nil;     /* initialize the pointer to a
                                    “safe” value before
                                    continuing... */
   myError = noErr;              /* initialize error to none */

      /* Here is the correct way to call OCECopyRString. This
       code allocates the destinationRString variable to the
      correct size before calling the OCECopyRString function. */

   destinationRString = (RString *)NewPtr(sizeof(RString));
   /* Check if memory allocation failed by calling MemError
      Toolbox function. */
   if (MemError() != noErr)
   {
      /* There was an error. Call your error handler. */
      DoErrorHandling(myError);
   }
   /* Otherwise the RString was allocated properly. */
   myError = OCECopyRString(sourceRString, destinationRString);
   if (myError != noErr)
   {
```

```
      /* There was an error. Call your error handler. */
      DoErrorHandling(myError);
   }
}
```

Copying Versus Duplicating AOCE Data Structures

There is a single AOCE duplication function, `OCEDuplicateRLI`; it is used to duplicate `RLI` data structures. The difference between copying and duplicating as performed by AOCE toolbox functions is subtle but important. In this context, ***copying*** is taking the contents of each field in the source structure and placing them in the corresponding field of the destination structure. This process includes all nested structures as well.

However, some AOCE data structures, such as `RLI` structures, contain fields that are pointers to other nested data structures. For this reason, it is possible to change the pointers in the destination structure so that they point to the corresponding data structures in the source structure. This process of copying the pointers to data structures and not the actual data structures themselves, is called ***duplicating*** the data structures. This distinction between copying and duplicating applies only to the AOCE utility functions, and not to other APIs.

There are advantages and disadvantages to duplicating a data structure as opposed to copying it, and you must decide when it is appropriate to use duplication or copying in your own code. The advantage of duplicating a data structure is that it is much faster and requires less code than copying because only a pointer must be moved instead of a whole data structure.

The disadvantage of duplication is that you must keep both the source and destination structures in memory until you have finished using them. Here is the reason why: When you duplicate a structure, the pointers in the destination structure change to point to the source structure. Thus, after you duplicate a data structure, there is really only one copy of the data, but that data is pointed to by both the source and destination structures.

Table 2-4 AOCE copying and duplicating functions and associated data structures

Copying Function Name	Data Structure Copied
`OCECopyRString`	`RString`
`OCECopyCreationID`	`CreationID`
`OCECopyPackedPathName`	`PackedPathName`
`OCECopyDirDiscriminator`	`DirDiscriminator`
`OCECopyRLI`	`RLI`
`OCEDuplicateRLI`	`RLI`
`OCECopyPackedRLI`	`PackedRLI`

continued

Table 2-4 AOCE copying and duplicating functions and associated data structures (continued)

Copying Function Name	Data Structure Copied
`OCECopyLocalRecordID`	`LocalRecordID`
`OCECopyShortRecordID`	`ShortRecordID`
`OCECopyRecordID`	`RecordID`
`OCECopyPackedRecordID`	`PackedRecordID`
`OCECopyPackedDSSpec`	`PackedDSSpec`

Allocating AOCE Strings of Nonstandard Sizes

Three standard AOCE string sizes are defined for you by the `RString`, `RString64`, and `RString32` structures. There are times, however, when you may wish to create an AOCE string of arbitrary size to store specialized data. Listing 2-4 shows how to accomplish this task. This example allocates an AOCE string that has a size of 23 bytes.

Listing 2-4 Allocating a string to store specialized data

```
ProtoRString *rstr;                    /* create a pointer to a
                                          ProtoRString struct */

rstr = NewPtr(23+sizeof(RStringHeader)); /* allocate memory for it
                                          including its header
                                          information */

if (rstr == nil)
{
   /* Then allocation has failed; not enough memory available.
      Put your error handling here. */
}
rstr->charSet = smRoman;            /* set script code to Roman */
rstr->length = 23;                  /* set the proper length */
```

Allocating a RecordID Structure of Maximum Size

When you allocate a new minimum-sized structure for the first time, memory is not automatically allocated for any of its fields except the header. There are times, however, when you may want to create a structure that has all of the memory for its fields allocated, thus ensuring that you have enough memory to hold a maximum-sized structure. For more information on minimum and maximum-sized AOCE structures, see "AOCE Data Structures of Maximum and Minimum Size" on page 2-3.

Listing 2-5 shows two functions, `MyAllocateMaxRID` and `MyDeallocateMaxRID`, which allocate and dispose of a maximum-sized `RecordID` structure. The `MyAllocateMaxRID` function uses the AOCE utility routine `OCESetCreationIDtoNULL` (page 2-54) to initialize the fields of a `CreationID` structure to `NULL` values. In addition, this function uses the Macintosh Toolbox routine `MemErr` to check for memory allocation errors.

Listing 2-5 Allocating and disposing of a maximum-sized `RecordID` structure

```
/* This function allocates a maximum-sized recordID structure*/

OSErr MyAllocateMaxRID(RecordID *rid)
{
   OSErr err;                        /* The error, if any, returned
                                        by AllocateMaxRID */
   PackedRLIPtr rli;                 /* Pointer to a packed RLI */
   RString *name;                    /* The record name */
   RString *type;                    /* The record type */

   rid->local.recordName = nil;      /* Initialize the record */
   rid->local.recordType = nil;      /* name, type, and rli to */
   rid->rli = nil;                   /* nil */

   /* Now allocate memory for a maximum-sized RString to hold
      the record name. */

   name = (RString*) NewPtr(sizeof(RString));

   err = MemError();
   if (err == noErr)
      {
      /* Now allocate space for the RString to hold the
         record type. */

      type = (RString*) NewPtr(sizeof(RString));

      err = MemError();
      if (err == noErr)
         {
         /* Finally, allocate the memory for the packed RLI. */

         rli = (PackedRLIPtr) NewPtr(sizeof(PackedRLI));

         err = MemError();
```

```
         if (err == noErr)
            {
            /* Now that all storage has been allocated, assign
               it to its proper location. */

            rid->local.recordName = name;
            rid->local.recordType = type;
            rid->rli = rli;

            /* Set the RLI's length field to its maximum size */

            rli->length = kRLIMaxBytes;

            /* Set the name and type RString's length fields to
               their maximum size. */

            name->length = kRStringMaxBytes;
            type->length = kRStringMaxBytes;

            /* Now initialize the creation ID by setting it to
               NULL. */

            OCESetCreationIDtoNull(&(rid->local.cid));

            }
         }
      }
   if (err != noErr)    /* if there was an error during memory */
                           allocation, dispose of the record ID
                           and return the error to the caller */
   {
      MyDeallocateMaxRID(rid);/* call function described next */
   }

   return err;
}

/* This function deallocates a record ID whose fields were
   allocated on the heap. */

void MyDeallocateMaxRID(RecordID *rid)
{
   DisposPtr((Ptr) rid->local.recordName);
```

```
   DisposPtr((Ptr) rid->local.recordType);
   DisposPtr((Ptr) rid->rli);
}
```

AOCE Utilities Reference

This section describes the data structures that are used throughout the various AOCE managers and packages and the utility functions that manipulate these data structures.

AOCE Data Structures

The data types described in this chapter are used throughout AOCE and are not confined to a particular manager or package.

AOCE String Structures

The AOCE string structures are used by AOCE functions in place of standard Pascal strings because AOCE strings can handle international character sets that may consist of 2 bytes per character and because AOCE strings include the script code for the character set of the data they contain. Standard Pascal strings use only 1 byte per character. All of the AOCE string structures consist of an `RStringHeader` field and a `body` field. The `RStringHeader` field contains information about the AOCE string, such as its character set and length, whereas the `body` field holds the actual string contents.

RStringHeader

The header is the portion of each AOCE string that defines the particular qualities that apply to the string's contents. Each header contains the field `charSet`, which is used to specify the character set, or script code, corresponding to the script you should use to interpret the AOCE string. A script code represents a writing system for a human language, such as Roman, Kanji, or Arabic, and the `charSet` field is the same as the script code used by the Script Manager to specify a particular script. See *Inside Macintosh: Text* for more information about script codes and international character sets, as well as for a listing of defined script code constants.

The header is defined as follows:

```
#define RStringHeader      \
   CharacterSet charSet;   \
   unsigned short dataLength;

   typedef short CharacterSet;
```

Field descriptions

`charSet`	The character set that applies to the text contained in the `RString`.
`datalength`	The length, in bytes, of the `body` field of the `RString` structure, not including the header. Note that for 2-byte character sets, such as Kanji, the number of characters in the `RString` structure is half the number of bytes in the `body` field.

RString

The `RString` structure is the basis for most strings in AOCE, as well as for other AOCE data types such as the `DirectoryName`, `AttributeType`, and `NetworkSpec` structures. The maximum number of bytes in an `RString` structure is defined by the constant `kRStringMaxBytes`, and the maximum number of characters in an `RString` structure is defined by the constant `kRStringMaxChars`.

Because the `RString` structure is of maximum size, it is already large enough to hold any other valid `RString` structure when you allocate it. For a minimum-sized AOCE string structure, see the `ProtoRString` type on page 2-22. The `RString` structure is defined as follows:

```
struct RString
{
   RStringHeader
   Byte  body[kRStringMaxBytes];
};

typedef struct RString RString;
```

Field descriptions

`RStringHeader`	A header (described on page 2-19) which defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the data in the `body` field of the `RString` structure.
`body`	An array containing the actual `RString` structure's characters. The array has a length of `kRStringMaxBytes` number of bytes and contains as many bytes of data as specified by the `dataLength` field of the header. The constant `kRStringMaxBytes` is equal to 256 bytes.

RString64

The `RString64` structure is identical to an `RString` structure, except that its maximum size is smaller. The `RString64` length is defined by the constant `kRString64Size.`

```
struct RString64
{
   RStringHeader
   Byte     body[kRString64Size];
};

typedef struct RString64 RString64;
```

Field descriptions

`RStringHeader` A header (described on page 2-19) which defines the character set information that applies to the text of the `RString64` structure and specifies the length, in bytes, of the data in the `body` field of the `RString64` structure.

`body` An array containing the actual `RString64` structure's characters. The array has a length of `kRString64Size` number of bytes and contains as many bytes of data as specified by the `dataLength` field of the header. The constant `kRString64Size` is equal to 64 bytes.

RString32

The `RString32` structure is identical to an `RString` structure, except that its maximum size is smaller. The `RString32` structure's length is defined by the constant `kRString32Size`.

```
struct RString32
{
   RStringHeader
   Byte     body[kRString32Size];
};

typedef struct RString32 RString32;
```

Field descriptions

`RStringHeader` A header (described on page 2-19) which defines the character set information that applies to the text of the `RString32` structure and specifies the length, in bytes, of the data in the `body` field of the `RString32` structure.

`body` An array containing the actual `RString32` structure's characters. The array has a length of `kRString32Size` number of bytes and contains as many bytes of data as specified by the `dataLength` field of the header. The constant `kRString32Size` is equal to 32 bytes.

ProtoRString

The `ProtoRString` is the only AOCE string structure of minimum size; it initially has no space allocated for the string contents. You should use a `ProtoRString` structure whenever you need to create an AOCE string of variable length.

```
struct ProtoRString
{
   RStringHeader
   /* Define the body of the ProtoRstring here. */
};

typedef struct ProtoRString ProtoRString;
```

Field descriptions

`RStringHeader` A header (described on page 2-19) which defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the data in the `body` field of the `RString` structure.

Note

The `ProtoRString` structure does not have a defined `body` field as do the other AOCE string structures. It is up to you to add a `body` field for the `ProtoRString` structure. See the section "Allocating AOCE Strings of Nonstandard Sizes" on page 2-16 for an example of how to do this. ◆

DirectoryName

A `DirectoryName` structure consists of a character set code, a length containing the number of bytes in the `body` field, and the data in the `body` field. A `DirectoryName` structure is identical to an `RString` structure, except that its maximum length is defined by the constant `kDirectoryNameMaxBytes` and its `body` field holds the name of a catalog (it is called a `DirectoryName` structure for historical reasons). You can typecast any `DirectoryName` structure to an `RString` structure and use the `RString` utility functions on it. The `RString` utility functions are described starting on page 2-45.

```
struct DirectoryName
{
   RStringHeader
   Byte     body[kDirectoryNameMaxBytes];
};

typedef struct DirectoryName DirectoryName;
```

Field descriptions

`RStringHeader` A header (described on page 2-19) which defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the data in the `body` field of the `RString` structure.

`body` An array of characters that contains the name of a catalog. This array can contain up to `kDirectoryNameMaxBytes` number of bytes and contains as many bytes of data as specified by the `dataLength` field of the header. The constant `kDirectoryNameMaxBytes` is equal to 32 bytes.

NetworkSpec

A `NetworkSpec` structure consists of a character set code, a length containing the number of bytes of data, and the data itself. A `NetworkSpec` structure is identical to an `RString` structure, except that its maximum length is defined by the constant `kNetworkSpecMaxBytes` and its `body` field is used to hold the name of a network. You can typecast any `NetworkSpec` structure to an `RString` structure and use any of the `RString` utility functions on it. The `RString` utility functions are described starting on page 2-45.

For an example of how some functions use the `NetworkSpec` structure, see the `DirGetLocalNetworkSpec` and `DirGetDNodeInfo` functions in the chapter "Catalog Manager" in this book.

```
struct NetworkSpec
{
   RStringHeader
   Byte  body[kNetworkSpecMaxBytes];
};

typedef struct NetworkSpec NetworkSpec;
```

The `RStringHeader`, described on page 2-19, defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the `body` field of the `RString` structure.

Field descriptions

`RStringHeader` A header (described on page 2-19) which defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the data in the `body` field of the `RString` structure.

`body`	An array of characters that contains the name of a network. This array can contain up to `kNetworkSpecMaxBytes` number of bytes and contains as many bytes of data as specified by the `dataLength` field of the header. The constant `kNetworkSpecMaxBytes` is equal to 32 bytes.

RStringKind

Some of the AOCE utility functions require a parameter of type `RStringKind` in addition to an AOCE string parameter. Based on the value of the parameter of type `RStringKind`, the routine determines how it will handle the `RString` structure. The `OCERelRString` (page 2-48), `OCEEqualRString` (page 2-50), and `OCEValidRString` (page 2-51) functions use the `RStringKind` data type. When you call one of these functions, you need to decide what value of the `RStringKind` type to use.

```
enum
{
   kOCEDirName =           0,
   kOCERecordOrDNodeName = 1,
   kOCERecordType =        2,
   kOCENetworkSpec =       3,
   kOCEAttrType =          4,
   kOCEGenericSensitive =  5,
   kOCEGenericInsensitive =6
};

typedef unsigned short RStringKind;
```

Field descriptions

`kOCEDirName`	The AOCE string is a `DirectoryName` structure containing a catalog name. For more information about the `DirectoryName` structure see page 2-22.
`kOCERecordOrDNodeName`	The AOCE string is a `recordName` structure containing a record name or a catalog node name. See the `LocalRecordId` structure on page 2-27 for the definition of the `recordName` structure.
`kOCERecordType`	The AOCE string is a `recordType` structure containing a record type. See the `LocalRecordId` structure on page 2-27 for more information on the `recordType` structure.
`kOCENetworkSpec`	The AOCE string is a `NetworkSpec` structure containing a network specification. See page 2-23 for more information on the `NetworkSpec` structure.

`kOCEAttrType` The AOCE string is an `AttributeType` structure containing an attribute type. For more information on the `AttributeType` structure see page 2-39.

`kOCEGenericSensitive`
The AOCE string is a generic AOCE string type that you should use when you want an AOCE utility routine to be both case-sensitive and sensitive to diacritical marks in its treatment of an `RString` structure (c ≠ C ≠ ç). Use this type for your own AOCE strings that will not be seen by a user.

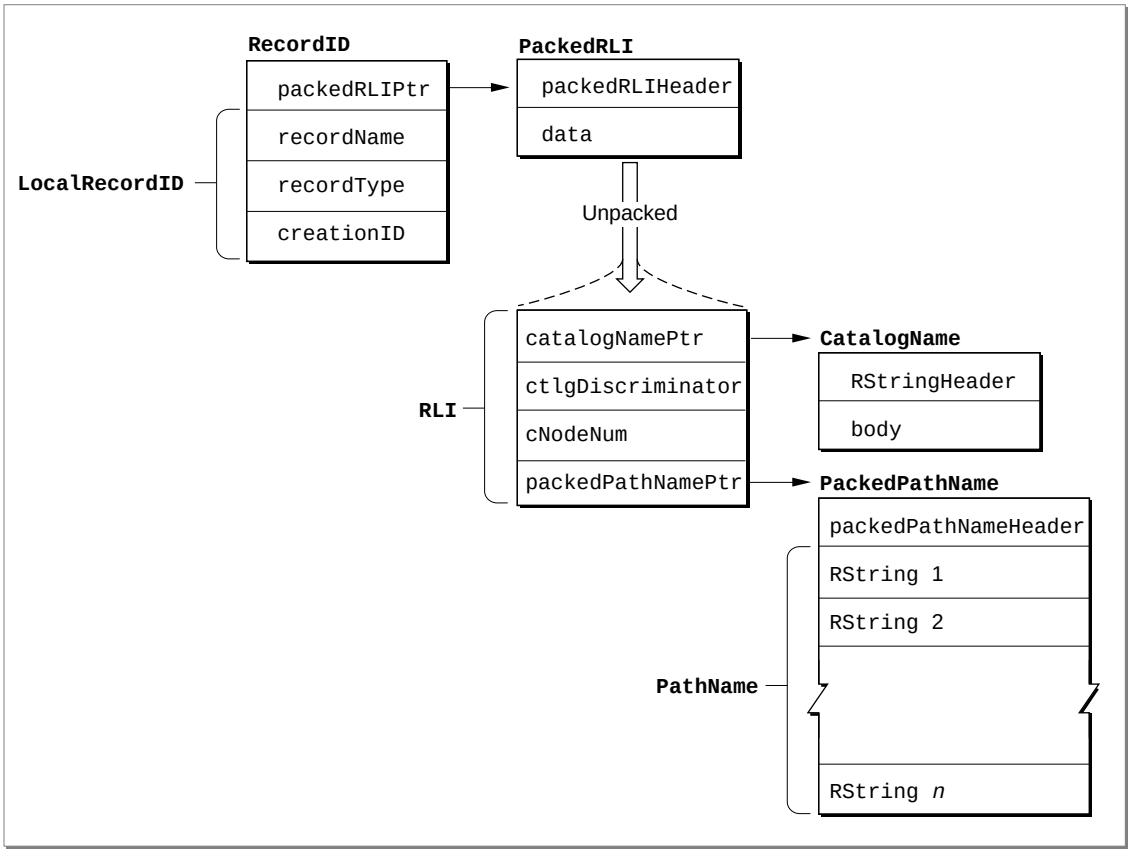
`kOCEGenericInsensitive`
The AOCE string is a generic AOCE string type that you should use when you want an AOCE utility routine to be neither case-sensitive nor sensitive to diacritical marks in its treatment of an `RString` structure (c = C = ç). Use this type for your own AOCE strings that will be seen by a user.

Note
You should use the `kOCEGenericSensitive` and `kOCEGenericInsensitive` `RStringKind` values when you use AOCE strings to hold data other than a catalog node name or the five derivative AOCE string structures (`DirectoryName`, `AttributeType`, `NetworkSpec`, `recordName`, and `recordType`). Do not use the `kOCEGenericSensitive` and `kOCEGenericInsensitive` `RStringKind` types with `DirectoryName`, `recordName`, `recordType`, `NetworkSpec`, or `AttributeType` structures or with catalog node names because this may cause the AOCE string to be treated incorrectly by the function you are calling. ◆

Record Identifier Structures

A record identifier structure uniquely identifies a record in an AOCE catalog. It consists of the name and discriminator value of the catalog, the catalog node number or the path information for the catalog node in which the record is located, and the record's name, type, and creation identifier. A record identifier is defined by the `RecordID` structure. Because the `RecordID` structure is composed of substructures (see Figure 2-1), many of which contain components of their own, the component structures of the `RecordID` structure are described first in this section.

Figure 2-1 The Record identifier structure



CreationID

The record creation identifier is defined by the `CreationID` structure and is used to uniquely identify a record within a PowerShare catalog or in a personal catalog. Some catalogs may not support the `CreationID` structure; they may rely on the uniqueness of a record's name and type to specify each record instead. The `CreationID` structure is a component of the `LocalRecordID` structure (page 2-27).

The fields of the `CreationID` structure are private to a catalog; you never need to know how to put data into a `CreationID` structure or how the data is represented inside the `CreationID` structure. Once you have allocated space for a new `CreationID` structure, you simply pass it into a function such as `DirAddRecord`, which fills the `CreationID` structure with the proper data for you. You then pass the `CreationID` structure along to other functions that require it, such as the `DirDeleteRecord` function. For more information on the `DirAddRecord` and `DirDeleteRecord` functions see the chapter "Catalog Manager" in this book.

AOCE defines two types of `CreationID` structures: the `CreationID` structure and the `AttributeCreationID` structure. These structures are identical but have different names to help distinguish the way in which they are used by various AOCE managers and functions. The `CreationID` structure is sometimes called the record `CreationID` structure to reinforce the idea that it is being used for a record, and not an attribute.

```
struct CreationID
{
   unsigned long  source;  /* private to a catalog.*/
   unsigned long  seq;     /* private to a catalog*/
};

typedef struct CreationID CreationID;

typedef CreationID AttributeCreationID;
```


LocalRecordID

A local record identifier uniquely identifies a record within a catalog. It contains the record's creation identifier, described in the previous section, and the record's name and type. The name and type can uniquely identify a record in an external catalog that does not support creation identifiers. The local record identifier is defined by the `LocalRecordID` structure.

The creation identifier field of the local record identifier is maintained by the catalog that contains the `LocalRecordID` structure. Whenever a record is created in a catalog that supports creation identifiers, the catalog assigns the record a new creation identifier that is unique within the catalog. This procedure prevents duplicate creation identifiers within the same catalog. Within a catalog that does not support creation identifiers, it is not possible to have two records with the same name and type, because the catalog uses the record's name and type to define a particular record uniquely.

The `LocalRecordID` structure is a component of the `RecordID` structure described on page 2-34, and is also a component of the `DirEnumSpec` structure, described in the chapter "Standard Catalog Package" in this book. See the `DirFindValue` function in the chapter "Catalog Manager" in this book for an example of a function that uses the `LocalRecordID` structure.

```
struct LocalRecordID
{
   CreationID  cid;              /* creation ID of the record */
   RStringPtr  recordName;       /* name of the record */
   RStringPtr  recordType;       /* type of record */
};
```

```
typedef struct LocalRecordID LocalRecordID;
typedef LocalRecordID *LocalRecordIDPtr;
```

Field descriptions

`cid`	The creation identifier of the record. If the creation identifier is not `NULL`, this number is unique within a catalog.
`recordName`	The name of the record. The name is not necessarily unique within a catalog.
`recordType`	The type of entity that the record represents. For example, the record could be of type `User`, `Group`, `LaserWriter`, and so forth. For a list of standard record types, see the `OCERecordTypeIndex` structure described next. The record type is not necessarily unique within a catalog.

OCERecordTypeIndex

The `OCERecordTypeIndex` is an enumerated list of the standard AOCE record types. You should use this list whenever you need to obtain a record type that has been defined by Apple Computer, Inc. All lowercase four-character combinations are reserved by Apple Computer, Inc., as well as all uppercase and lowercase combinations of the sequence 'AOCE'. To get a specific record type, call the `OCEGetIndRecordType` function and pass it the proper index constant from the `OCERecordTypeIndex` enumerated list. The `OCEGetIndRecordType` function returns a pointer to an `RString` structure that contains the proper record type corresponding to the index entry you supplied. See page 2-85 for the complete description of the `OCEGetIndRecordType` function.

```
enum /* OCERecordTypeIndex */
{
   kUserRecTypeNum =          1,    /* "User" */
   kGroupRecTypeNum =         2,    /* "Group" */
   kMnMRecTypeNum =           3,    /* "AppleMail™ M&M" */
   kMnMForwarderRecTypeNum =  4,    /* "AppleMail™ Fwdr" */
   kNetworkSpecRecTypeNum =   5,    /* "NetworkSpec" */
   kADAPServerRecTypeNum =    6,    /* "PowerShare Server" */
   kADAPDNodeRecTypeNum =     7,    /* "PowerShare DNode" */
   kADAPDNodeRepRecTypeNum =  8,    /* "PowerShare DNode Rep" */
   kServerSetupRecTypeNum =   9,    /* "Server Setup" */
   kDirectoryRecTypeNum =     10,   /* "Catalog" */
   kDNodeRecTypeNum =         11,   /* "DNode" */
   kSetupRecTypeNum =         12,   /* "Setup" */
   kMSAMRecTypeNum =          13,   /* "MSAM" */
   kDSAMRecTypeNum =          14,   /* "CSAM" */
   kAttributeValueRecTypeNum =15,   /* "Attribute Value" */
```

```
   kBusinessCardRecTypeNum =  16,   /* "Business Card" */
   kMailServiceRecTypeNum =   17,   /* "Mail Service" */
   kCombinedRecTypeNum =      18,   /* "Combined" */
   kOtherServiceRecTypeNum =  19,   /* "Other Service" */
   kAFPServiceRecTypeNum =    20    /* "Other Service afps" */
};

typedef unsigned short OCERecordTypeIndex;
```

In addition to the `OCERecordTypeIndex` values defined above, there are three more record type definitions:

```
#define kFirstOCERecTypeNum kUserRecTypeNum
            /* first standard AOCE record type */

#define kLastOCERecTypeNum kAFPServiceRecTypeNum
            /* last standard AOCE record type */

#define kNumOCERecTypes (kLastOCERecTypeNum -
                         kFirstOCERecTypeNum + 1)/* total number of
                                                    standard AOCE
                                                    record types */
```

You can use these three constants to enumerate all the standard AOCE record types.

PackedPathName

The `PackedPathName` structure contains the names of all of the catalog nodes in the path from the catalog node in which a record resides, to the root catalog node in the AOCE catalog tree. A `PackedPathName` structure is an array of `RString` structures, with each component `RString` structure containing the name of a catalog node on the path. You create a `PackedPathName` structure from an array of `RString` structures by using the `OCEPackPathName` function (page 2-60). You can also unpack a `PackedPathName` structure into its `RString` component parts by using the `OCEUnpackPathName` function (page 2-58). The maximum size of an entire packed pathname is defined by the constant `kPathNameMaxBytes`.

The `PackedPathName` structure's format is private, so you must always use the `OCEPackPathName` and `OCEUnpackPathName` functions to pack and unpack these structures. Do not assume you know the format of `PackedPathName` structures.

The `PackedPathName` structure is a component of the record location information structure (page 2-32). In addition, the AOCE Catalog Manager uses the packed pathname structure in various functions such as `DirMapDNodeToPathName` and `DirMapPathNameToDNode`. For information on these functions, see the chapter "Catalog Manager" in this book.

```
struct PackedPathName
{
   unsigned short  dataLength;       /* number of bytes in data
                                        field */
   Byte         data[kPathNameMaxBytes - sizeof (unsigned short)];
};

typedef struct PackedPathName PackedPathName;
```

Field descriptions

`dataLength` The number of bytes in the data field. This does not include the bytes in the dataLength field itself.

`data` A packed array containing the names of all of the catalog nodes in the path from the catalog node in which the record resides, to the catalog root node. Each of the names in the array is an `RString` structure.

ProtoPackedPathName

The `ProtoPackedPathName` structure is a minimum-sized structure. It is equivalent to a `PackedPathName` structure without a data field. You should use this data type whenever you need to create a `PackedPathName` structure of variable length.

```
struct ProtoPackedPathName {
   unsigned short dataLength;
/* Followed by data */
};

typedef struct ProtoPackedPathName ProtoPackedPathName;
```

Field descriptions

`dataLength` The length of the data field of the PackedPathName structure.

Note

You must create the data portion of the `ProtoPackedPathName` structure yourself. Since this is a minimum-sized structure, it initially has no `data` field, and hence no memory is allocated for any contents. See the section "Allocating AOCE Strings of Nonstandard Sizes" on page 2-16 for an example of how to allocate memory for a minimum-sized structure. ◆

DirDiscriminator

A catalog discriminator is defined by a `DirDiscriminator` structure and is used to differentiate between two or more catalogs that have the same name, as the combination of a catalog name and a `DirDiscriminator` structure uniquely identify a catalog. The `DirDiscriminator` structure contains two fields which are set by the catalog. An application does not need to set or change these fields. If you are creating a catalog server access module, you need to read the chapter "Catalog Service Access Modules" in *Inside Macintosh: AOCE Service Access Modules* for information on how to modify the fields of a `DirDiscriminator` structure.

In addition to being a component of the record location information structure, described next, the `DirDiscriminator` structure is used by several of the AOCE Catalog Manager functions. You also use a `DirDiscriminator` structure when you provide callback functions to such functions as `DirEnumerateDirectoriesParse` and `DirNetSearchADAPDirectoriesParse`. See the chapter "Catalog Manager" in this book for more information on these two functions.

```
struct DirDiscriminator {
   OCEDirectoryKind  signature;     /* type of a catalog */
   unsigned long     misc;          /* private to catalog */
};

typedef struct DirDiscriminator DirDiscriminator;
```

Field descriptions

`signature` Defined by the catalog provider. It may be, but is not required to be, the same as the application's signature. Apple Computer, Inc. has defined the following values for this field. Developers of catalog service access modules may define additional values.

`kDirAllKinds = 0`

`kDirADAPKind` = `'adap'`

`kDirPersonalDirectoryKind` = `'pdir'`

`kDirDSAMKind = 'dsam'`

`misc` Defined by the catalog provider. A catalog service access module may use it to distinguish between different catalogs that it supports. See the chapter "Catalog Service Access Modules" in *Inside Macintosh: AOCE Service Access Modules* for more information on this field.

RLI

The record location information structure identifies the catalog and catalog node in which a record resides. The record location information is defined by the `RLI` data type. The `RLI` structure is the unpacked form of the `PackedRLI` data structure, described next.

```
typedef unsigned long   DNodeNum;

struct RLI {
   DirectoryNamePtr     directoryName;
   DirDiscriminator     discriminator;
   DNodeNum             dNodeNumber;
   PackedPathNamePtr    path;
};

typedef struct RLI RLI;
typedef RLI *RLIPtr;
```

Field descriptions

`directoryName` A pointer to the name of the catalog in which the record resides. The maximum number of bytes in a catalog name is defined by the constant `kDirectoryNameMaxBytes`.

`discriminator` A value that allows you to distinguish between two or more catalogs that have the same name.

`dNodeNumber` A value that uniquely identifies the catalog node in which the record resides. Set this field to 0 or to `kNULLDNodeNumber` if you are using the `path` field to identify the catalog node.

`path` A pointer to a buffer that contains the names of all of the catalog nodes on the path from the catalog node in which the record resides, to the catalog root node. You should set this field to `nil` if you are using the `dNodeNumber` field to identify the catalog node.

The `directoryName` and `discriminator` fields of the `RLI` structure specify the catalog. The last two fields of the `RLI` structure, the `dNodeNumber` and `path` fields, specify a catalog node within the catalog specified by the `directoryName` and `discriminator` fields. For PowerShare catalogs, you must specify the catalog node by either a catalog node number or by a pathname, but not both.

Some catalogs may allow you to specify a catalog node using a partial pathname. A partial pathname is a combination of values in the `dNodeNumber` and `path` fields. To assure compatibility with all catalogs, you need to call the `DirGetDirectoryInfo` function to find out if the catalog supports the use of partial pathnames before providing a partial pathname to the catalog. If a catalog supports partial pathnames, you must set both the `dNodeNumber` and `path` fields to meaningful values, because both fields are used. If this is the case, and your application does not support partial pathnames, you should set either the `dNodeNumber` field to 0 or the `path` field to `nil`.

PackedRLI

The record location information in its packed form is defined by the `PackedRLI` data type. Use the `OCEPackRLI` function (page 2-71) to create a `PackedRLI` structure from an `RLI` structure or its component parts. Use the `OCEUnpackRLI` function (page 2-72) to unpack a `PackedRLI` structure into its component parts. The order of the data within a `PackedRLI` structure is private, so you must use the utility functions when creating and unpacking `PackedRLI` structures. This is the only way to be sure that the data will be in the correct format.

In addition to being a component of the `RecordID` data structure, described on page 2-34, the `PackedRLI` structure is used by several of the AOCE Catalog Manager functions.

```
#define kRLIMaxBytes (sizeof (RString) + \
                      sizeof (DirDiscriminator) + \
                      sizeof (DNodeNum) + kPathNameMaxBytes)
```

The constant `kRLIMaxBytes` is the maximum number of bytes that can be stored in the `data` field of a `PackedRLI` structure. This is large enough to hold the sum of `RString`, `DirDiscriminator`, and `DNodeNum` structures plus a maximum-length pathname.

```
struct PackedRLI {
   unsigned short dataLength;          /* length of data field */
   Byte           data[kRLIMaxBytes];  /* packed record
                                          location info */
};

typedef struct PackedRLI PackedRLI;
typedef PackedRLI *PacedPLIPtr;
```

Field descriptions

`dataLength` The number of bytes in the `data` field of the `PackedRLI` structure. It does not include the number of bytes in the `dataLength` parameter itself.

`data` A packed array of characters that contains the catalog name, the catalog discriminator, and the catalog node number or a pathname.

ProtoPackedRLI

The `ProtoPackedRLI` structure is a minimum-sized structure. It is equivalent to a `PackedRLI` structure without a data field. You should use this data type whenever you need to create a `PackedRLI` structure of variable length.

```
struct ProtoPackedRLI {
   unsigned short dataLength;          /* length of data */
/* Followed by data */
};

typedef struct ProtoPackedRLI ProtoPackedRLI;
typedef ProtoPackedRLI *ProtoPackedRLIPtr;
```

Field descriptions

`dataLength` The length of the `data` field of the `PackedRLI` structure.

Note

You must create the data portion of the `ProtoPackedRLI` structure yourself. Because this is a minimum-sized structure, it initially has no `data` field, and thus no memory is allocated for any contents. See the section "Allocating AOCE Strings of Nonstandard Sizes" on page 2-16 for an example of allocating memory for a minimum-sized structure. ◆

RecordID

Each record in an AOCE catalog is described by a `RecordID` structure. A `RecordID` structure consists of two parts: a local record identifier and a packed record location information structure. The local record identifier uniquely defines the record within its catalog. The packed record location information structure identifies the catalog and catalog node in which the record resides.

```
struct RecordID {
   PackedRLIPtr      rli;     /* identifies record's catalog
                                 and dNode */
   LocalRecordID     local;   /* identifies record within
                                 its dNode */
};

typedef struct RecordID RecordID;
typedef RecordID *RecordIDPtr;
```

Field descriptions

`rli` A pointer to a `PackedRLI` structure that identifies the catalog and the specific catalog node in which the record resides.

`local` A `LocalRecordID` structure that uniquely identifies the record within its catalog.

PackedRecordID

A packed record identifier is the packed form of a `RecordID` structure and is defined by the `PackedRecordID` structure. The packed form of the `RecordID` structure is useful when you wish to store data or transmit it because the `PackedRecordID` structure is a single block of data, rather than a structure containing pointers into other structures as the `RecordID` structure is. You use the `OCEPackRecordID` function (page 2-90) to create a `PackedRecordID` structure from a `RecordID` structure, and you use the `OCEUnpackRecordID` function (page 2-91) to convert a `PackedRecordID` structure into an unpacked `RecordID` structure.

```
#define kPackedRecordIDMaxBytes (kPathNameMaxBytes + \
         sizeof (DNodeNum) + sizeof (DirDiscriminator) + \
         sizeof (CreationID) + (3 * sizeof (RString)))
```

The constant `kPackedRecordIDMaxBytes` defines the maximum number of bytes that can be stored in the `data` field of a `PackedRecordID` structure.

```
struct PackedRecordID {
   unsigned short dataLength; /* length of data field
                                 in PackedRecordID */
   Byte     data[kPackedRecordIDMaxBytes];/* packed record ID */
};

typedef struct PackedRecordID PackedRecordID;
```

Field descriptions

`dataLength` The size of the `data` field of the `PackedRecordID` structure. It does not include the length of the `dataLength` parameter itself.

`data` An array containing the `RecordID` data.

ShortRecordID

A short record identifier structure is similar to a record identifier, except that it does not contain the `recordName` and `recordType` fields. For more information on record location information structures see page 2-32.

```
struct ShortRecordID
{
   PackedRLIPtr rli;
   CreationID cid;
};

typedef struct ShortRecordID ShortRecordID;
```

Field descriptions

`rli`	A pointer to a packed record location information structure.
`cid`	A pointer to a creation identifier structure.

Catalog Services Specification

The catalog services specification structures are used throughout AOCE for performing various tasks such as getting and setting access controls for records, obtaining the individual members of a group record that the user has selected, computing the size of a record currently selected by the user, specifying message addresses, and so forth. The catalog services specification is defined by the `DSSpec` structure and its packed form by the `PackedDSspec` structure. Other forms of the `DSSpec` structure include the `OCERecipient` and the packed form, `OCEPackedRecipient`, which are defined in the chapter "Interprogram Messaging Manager" in this book.

In addition to the above uses, you can also use the catalog services specification to hold your own types of data that may not have a specified size. In this case, use the `ProtoPackedDSspec` structure.

DSSpec

The catalog services specification structure is defined by the `DSSpec` data type. A `DSSpec` structure contains a pointer to a `RecordID` structure, plus additional information such as an extension type, extension size, and extension value. When you supply a `DSSpec` structure to a routine, you must provide a pointer to a record identifier in its `entitySpecifier` field. The other fields are optional, depending upon what data the `DSSpec` structure is being used to hold. For example, if the `DSSpec` structure has no extension, then it can represent either the root of all catalogs, a single catalog, a catalog node, or a record. If the `DSSpec` structure has an extension, then the `extensionType`, `extensionSize`, and `extensionValue` fields must contain valid values for the particular extension type. For more information on extension types and their allowable values, see the `OCEValidDSSpec` function on page 2-102 and the `OCEGetDSSpecInfo` function on page 2-103.

One of the uses for the `DSSpec` structure is to specify access controls for a catalog node, record, or attribute type that supports access controls. The way that you accomplish this for PowerShare catalogs, for example, is to obtain a `DSSpec` structure by calling the `OCEGetAccessControlDSSpec` function. This function returns a pointer to a `DSSpec` structure based on the information you supply when you call the function. You can then use the `DSSpec` structure with access control functions such as `DirGetDNodeAccessControlGet`. For information on access control functions, see the section "Getting Access Controls" in the chapter "Catalog Manager" in this book.

```
struct DSSpec {
   RecordID       *entitySpecifier;
   OSType         extensionType;
   unsigned short extensionSize;
   Ptr            extensionValue;
};

typedef struct DSSpec DSSpec;
typedef DSSpec *DSSpecPtr;
```

Field descriptions

`entitySpecifier`
A pointer to a `RecordID` structure that contains the record information pertaining to the `DSSpec`. If the extension type is not `'entn'`, the contents of this field determine whether the `DSSpec` structure represents a catalog, a catalog node, a record, or the root of all catalogs.

`extensionType` The extension type of the `DSSpec` structure, if any. If the extension type is `'entn'` then the `DSSpec` has an extension. To determine whether a `DSSpec` structure has an extension type or not, you call the `OCEGetDSSpecInfo` function (page 2-103).

`extensionSize` The size, in bytes, of the extension (if any).

`extensionValue`
A pointer to the data of the extension.

PackedDSSpec

The `PackedDSSpec` structure is the packed form of the `DSSpec` structure. The `PackedDSSpec` structures are used by AOCE in various functions. For example, the `SDPGetPanelSelection` function uses a `PackedDSSpec` structure to indicate the record that the user has selected. Another use of the `PackedDSSpec` structure is as a component of an `Attribute` structure. If an attribute value has a `tag` field set to the value `typePackedDSSpec`, then the attribute contains data of type `PackedDSSpec`.

You can use the functions `OCEUnpackDSSpec` (page 2-98) and `OCEPackDSSpec` (page 2-97) to convert between the packed and unpacked forms of the `DSSpec` structure.

Note
The `PackedDSSpec` is not a maximum-sized structure. When you allocate a `PackedDSSpec` structure it will hold any valid packed `RecordID` structure, but not necessarily any additional extension data. ◆

```
#define kPackedDSSpecMaxBytes(sizeof (PackedRecordID) + \
   sizeof (OSType) + sizeof (unsigned short))
```

The constant `kPackedDSSpecMaxBytes` is the maximum size in bytes that can be stored in the `data` field of a `PackedDSSpec` structure.

```
struct PackedDSSpec {
      unsigned short    dataLength;/* length of data field */
      Byte              data[kPackedDSSpecMaxBytes];
};
```

Field descriptions

`dataLength` The length of the data field of the `PackedDSSpec` structure. This does not include the bytes in the `dataLength` field itself.

`data` An array containing the actual contents of the `PackedDSSpec`. The size of the `data` array is equal to `kPackedDSSpecMaxBytes` bytes.

```
typedef struct PackedDSSpec PackedDSSpec;
```

ProtoPackedDSSpec

The `ProtoPackedDSSpec` structure is a minimum-sized structure. It is equivalent to a `PackedDSSpec` structure without a data field. You should use this data type whenever you need to create a variable length packed `DSSpec` structure.

```
struct ProtoPackedDSSpec {
   unsigned short    dataLength;/* length of data field */
/* Followed by data */
};

typedef struct ProtoPackedDSSpec ProtoPackedDSSpec;
typedef ProtoPackedDSSpec *ProtoPackedDSSpecPtr;
```

Field descriptions

`dataLength` The length of the data field of the `PackedDSSpec` structure.

Note

You must create the data portion of the `ProtoPackedDSSpec` structure yourself. Since this is a minimum-sized structure, it initially has no `data` field and hence no memory is allocated for any contents. ◆

Attribute Structures

The attribute structures are used in AOCE to provide access to a record's contents, as well as to determine what type of data is stored in a record. The three main attribute structures are `Attribute`, `AttributeType`, and `AttributeValue`. The `Attribute` structure contains `AttributeValue` and `AttributeType` structures as components.

The AttributeValue structure is described on page 2-42. The AttributeType structure is a derivative of the RString structure (page 2-20) and is described on page 2-39.

Attributes

In AOCE, all information in a record is stored as attribute values of the record. An attribute can hold any type of data, and it is defined by the Attribute structure. Each Attribute structure contains an AttributeType, AttributeCreationID, and AttributeValue component. Certain types of attributes have been reserved by Apple Computer, Inc., but you can create other types as needed. The Attribute structure provides you with all the information you need to manipulate an attribute value. Because an attribute value may contain vastly different types of data depending upon its type, it is vital that you determine the type of attribute before attempting to manipulate or use its value.

Because the Attribute structure is composed of several substructures such as AttributeValue, which may contain structures of their own, the Attribute structure is described last in this section, after its component structures.

AttributeType

An attribute type is a component of the Attribute structure and is used to indicate what kind of information is stored in the value field of an Attribute structure. For a complete description of the Attribute and AttributeValue structures, see page 2-44 and page 2-42 respectively. You can define your own attribute types or use a standard attribute type. For a list of standard attribute types and their data formats see the description of OCEAttributeTypeIndex, next.

An attribute type consists of a character set code, a length containing the number of bytes in the body field, and the data in the body field. An AttributeType structure is identical to an RString structure, except that its maximum length is defined by the constant kAttributeTypeMaxBytes and its body field specifies the type of a given attribute. Attribute types must be larger than 0 bytes; AOCE does not allow NULL attribute types. You can typecast any AttributeType structure to an RString structure and use the RString utility functions on it. The RString utility functions are described in "AOCE String Functions" beginning on page 2-45.

In addition to being a component of an Attribute structure, the AttributeType structure is used by several of the AOCE Catalog Manager functions. In particular, the callback functions you create for the DirLookupParse and DirEnumerateAttributeTypesGet functions take an attribute type as an input. See the chapter "Catalog Manager" in this book for more information on these functions.

An attribute type is defined as follows:

```
struct AttributeType
{
   RStringHeader
   Byte  body[kAttributeTypeMaxBytes];
};

typedef struct AttributeType AttributeType;
typedef AttributeType *AttributeTypePtr;
```

The `RStringHeader`, described on page 2-19, defines the character set information that applies to the text of the `RString` structure and specifies the length, in bytes, of the `body` field of the `RString` structure.

Field descriptions

`body` An array of characters that contains the name of an attribute type. The maximum length of an attribute type is defined by the constant `kAttributeTypeMaxBytes`, and is equal to 32 bytes.

OCEAttributeTypeIndex

You should use the attribute type index whenever you need to obtain a standard attribute type. To do this, you call the `OCEGetIndAttributeType` function (page 2-94) with the proper value from the `OCEAttributeTypeIndex` list. The `OCEGetIndAttributeType` function returns a pointer to an `RString` structure containing the standard attribute type based on the index value you supplied.

All lowercase four-character combinations are reserved by Apple Computer, Inc., as are all uppercase and lowercase combinations of the sequence 'AOCE'.

```
#define  kMemberAttrTypeNum         1001  /* "Member" */
#define  kAdminsAttrTypeNum         1002  /* "Administrators" */
#define  kMailSlotsAttrTypeNum      1003  /* "mailslots" */
#define  kPrefMailAttrTypeNum       1004  /* "pref mailslot" */
#define  kAddressAttrTypeNum        1005  /* "Address" */
#define  kPictureAttrTypeNum        1006  /* "Picture" */
#define  kAuthKeyAttrTypeNum        1007  /* "auth key" */
#define  kTelephoneAttrTypeNum      1008  /* "Telephone" */
#define  kNBPNameAttrTypeNum        1009  /* "NBP Name" */
#define  kQMappingAttrTypeNum       1010  /* "ForwarderQMap" */
#define  kDialupSlotAttrTypeNum     1011  /* "DialupSlotInfo" */
#define  kHomeNetAttrTypeNum        1012  /* "Home Internet" */
#define  kCoResAttrTypeNum          1013  /* "Co-resident M&M" */
#define  kFwdrLocalAttrTypeNum      1014  /* "FwdrLocalRecord" */
```

```
#define  kConnectAttrTypeNum         1015  /* "Connected To" */
#define  kForeignAttrTypeNum         1016  /* "Foreign RLIs" */
#define  kOwnersAttrTypeNum          1017  /* "Owners" */
#define  kReadListAttrTypeNum        1018  /* "ReadList" */
#define  kWriteListAttrTypeNum       1019  /* "WriteList" */
#define  kDescriptorAttrTypeNum      1020  /* "Descriptor" */
#define  kCertificateAttrTypeNu      1021  /* "Certificate" */
#define  kMsgQsAttrTypeNum           1022  /* "MessageQs" */
#define  kPrefMsgQAttrTypeNum        1023  /* "PrefMessageQ" */
#define  kMasterPFAttrTypeNum        1024  /* "MasterPF" */
#define  kMasterNetSpecAttrTypeNum   1025  /* "MasterNetSpec" */
#define  kServersOfAttrTypeNum       1026  /* "Servers Of" */
#define  kParentCIDAttrTypeNum       1027  /* "Parent CID" */
#define  kNetworkSpecAttrTypeNum     1028  /* "NetworkSpec" */
#define  kLocationAttrTypeNum        1029  /* "Location" */
#define  kTimeSvrTypeAttrTypeNum     1030  /* "TimeServer Type" */
#define  kUpdateTimerAttrTypeNum     1031  /* "Update Timer" */
#define  kShadowsOfAttrTypeNum       1032  /* "Shadows Of" */
#define  kShadowServerAttrTypeNum    1033  /* "Shadow Server" */
#define  kTBSetupAttrTypeNum         1034  /* "TB Setup" */
#define  kMailSetupAttrTypeNum       1035  /* "Mail Setup" */
#define  kSlotIDAttrTypeNum          1036  /* "SlotID" */
#define  kGatewayFileIDAttrTypeNum   1037  /* "Gateway FileID" */
#define  kMailServiceAttrTypeNum     1038  /* "Mail Service" */
#define  kStdSlotInfoAttrTypeNum     1039  /* "Std Slot Info" */
#define  kAssoDirectoryAttrTypeNum   1040  /* "Asso. Catalog" */
#define  kDirectoryAttrTypeNum       1041  /* "Catalog" */
#define  kDirectoriesAttrTypeNum     1042  /* "Catalogs" */
#define  kSFlagsAttrTypeNum          1043  /* "SFlags" */
#define  kLocalNameAttrTypeNum       1044  /* "Local Name" */
#define  kLocalKeyAttrTypeNum        1045  /* "Local Key" */
#define  kDirUserRIDAttrTypeNum      1046  /* "Dir User RID" */
#define  kDirUserKeyAttrTypeNum      1047  /* "Dir User Key" */
#define  kDirNativeNameAttrTypeNum   1048  /* "Dir Native Name" */
#define  kCommentAttrTypeNum         1049  /* "Comment" */
#define  kRealNameAttrTypeNum        1050  /* "Real Name" */
#define  kPrivateDataAttrTypeNum     1051  /* "Private Data" */
#define  kDirTypeAttrTypeNum         1052  /* "Catalog Type" */
#define  kDSAMFileAliasAttrTypeNum   1053  /* "CSAM File Alias" */
#define  kCanAddressToAttrTypeNum    1054  /* "Can Address To" */
#define  kDiscriminatorAttrTypeNum   1055  /* "Discriminator" */
#define  kAliasAttrTypeNum           1056  /* "Alias" */
#define  kParentMSAMAttrTypeNum      1057  /* "Parent MSAM" */
```

```
#define  kParentDSAMAttrTypeNum     1058  /* "Parent CSAM" */
#define  kSlotAttrTypeNum           1059  /* "Slot" */
#define  kAssoMailServiceAttrTypeNum1060  /* "Asso. Mail
                                             Service" */
#define  kFakeAttrTypeNum           1061  /* "Fake" */
#define  kInheritSysAdminAttrTypeNum1062  /* "Inherit
                                             SysAdministrators" */
#define  kPreferredPDAttrTypeNum    1063  /* "Preferred PD" */
#define  kLastLoginAttrTypeNum      1064  /* "Last Login" */
#define  kMailerAOMStateAttrTypeNum 1065  /* "Mailer AOM State" */
#define  kMailerSendOptionsAttrTypeNum \
                                    1066  /* "Mailer Send
                                             Options" */
#define  kJoinedAttrTypeNum         1067  /* "Joined" */
#define  kUnconfiguredAttrTypeNum   1068  /* "Unconfigured" */
#define  kVersionAttrTypeNum        1069  /* "Version" */
#define  kLocationNamesAttrTypeNum  1070  /* "Location Names" */
#define  kActiveAttrTypeNum         1071  /* "Active" */
#define  kDeleteRequestedAttrTypeNum
                                    1072  /* "Delete Requested" */
#define  kGatewayTypeAttrTypeNum    1073  /* "Gateway Type" */
```

In addition, Apple Computer, Inc., has defined three other attribute type constants to simplify the task of enumerating the standard attribute types.

```
typedef unsigned short OCEAttributeTypeIndex;

#define kFirstOCEAttrTypeNum kMemberAttrTypeNum
/* the first standard attribute type */

#define kLastOCEAttrTypeNum kGatewayTypeAttrTypeNum
/* the last standard attribute type */

#define kNumOCEAttrTypes (kLastOCEAttrTypeNum -
                          kFirstOCEAttrTypeNum + 1)
                          /* the total number of attributes */
```

AttributeValue

The `AttributeValue` structure consists of a `tag` field that indicates the format of the attribute value, a `datalength` field specifying the number of bytes contained in the attribute value, and a pointer to the attribute value data itself. Apple Computer, Inc. has reserved tags for attribute values that consist of `RString` and `PackedDSSpec`

structures, as well as for an unspecified sequence of bytes. You can also define your own tags to specify the attribute value formats that you have created.

```
typedef DescType AttributeTag;/* same type used in AppleEvents */

enum {
   typeRString       =  'rstr',
   typePackedDSSpec  =  'dspc',
   typeBinary        =  'bnry'
};
```

Constant descriptions

`typeRString` The attribute value is an `RString` structure.

`typePackedDSSpec` The attribute value is a `PackedDSSpec` structure.

`typeBinary` The attribute value is a sequence of bytes not defined by a formal structure.

```
struct AttributeValue {
   AttributeTag   tag;        /* format of attribute value */
   unsigned long  dataLength; /* # of bytes in attribute value */
   Ptr            bytes;      /* points to attribute value data */
};

typedef struct AttributeValue AttributeValue;
typedef AttributeValue *AttributeValuePtr;
```

Field descriptions

`tag` A value that indicates the format of the attribute value contained in the `bytes` field. If the `tag` field is set to `'rstr'`, the attribute value is considered to be an `RString` type.

If the attribute value is an `RString` structure, then the maximum size of the `body` field of the `RString` structure is (`kAttrValueMaxBytes - sizeof(ProtoRString)`) bytes.

If the attribute value is a `DSSpec` structure, then the maximum amount of data that can be stored in the `DSSpec` structure is (`kAttrValueMaxBytes - sizeof(ProtoPackedDSSpec)`) bytes.

The `tag` field can also contain a value defined by you that specifies the format of the attribute value.

Apple's PowerShare catalogs and personal catalogs restrict attribute values to a maximum size of `kAttrValueMaxBytes` bytes. If the `tag` field is set to `'dspc'`, the attribute value is a `PackedDSSpec` type.

`dataLength`	The number of bytes in the buffer pointed to by the `bytes` field. If the `tag` field is equal to `'rstr'` or `'dspc'`, then this length also includes the size of the `dataLength` field of the `DSSpec` structure or the `RStringHeader` of the `RString` structure.
`bytes`	A pointer to a buffer that contains the attribute value. You must provide this buffer. The constant `kAttrValueMaxBytes` defines the maximum size of any attribute value.

Attribute

The `Attribute` structure completely defines an attribute value by specifying its attribute type, attribute creation identifier, attribute tag, and the attribute value.

```
typedef CreationID   AttributeCreationID;

struct Attribute {
   AttributeType        attributeType; /* type of the attribute */
   AttributeCreationID  cid;           /* the creationID of the
                                          attribute */
   AttributeValue       value;         /* the attribute value */
};

typedef struct Attribute Attribute;
```

Field descriptions

`attributeType`	The attribute type. Apple Computer, Inc. has reserved all attribute types that are four-letter lowercase combinations, as well as any uppercase and lowercase combination of the letters 'AOCE'. A complete list of reserved attribute types can be found on page 2-40.
`cid`	The attribute creation identifier that uniquely defines the attribute value within the record. The `AttributeCreationID` structure has the same definition as the `CreationID` structure (see page 2-26).
`value`	The data for the attribute.

AOCE Utility Functions

The AOCE utility functions make it easier to manipulate the AOCE data structures. These functions perform various tasks such as comparison, duplication, creation, and conversion of structures. To call any of the functions described here from assembly language, you need to perform the following actions:

1. Leave space on the stack for the function result, if any.
2. Push the parameters on the stack using Pascal calling convention. This means that `parameter1` is pushed first, `parameter2` is pushed second, and so forth.

3. Place the routine selector in register D0.
4. Call the `__OCEUtils` trap macro.

AOCE String Functions

The AOCE string functions described in this section facilitate the creation, duplication, and conversion of AOCE strings.

OCECopyRString

The `OCECopyRString` function copies one AOCE string into another AOCE string.

```
pascal OSErr OCECopyRString (const RString *str1, RString *str2,
                    unsigned short str2Length);
```

`str1` A pointer to the source AOCE string that you want to copy from. You must provide this structure.

`str2` A pointer to the destination AOCE string that you want to copy to. You must provide this structure.

`str2Length`
The length of the destination AOCE string, not including the header information.

DESCRIPTION

The `OCECopyRString` function copies the contents of the source AOCE string into the destination AOCE string. If the destination string is not large enough to hold the contents of the source string, then the `OCECopyRString` function returns a memory-full error. You obtain the proper size needed for the destination AOCE string from the value contained in the `RStringHeader` field of the source AOCE string. Once you obtain this value, you can then use it to allocate a destination AOCE string of the proper size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0308

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory to copy the source string into the destination string, or the destination string is not large enough to hold the source string

SEE ALSO

The `RString` structure is described on page 2-20.

OCECToRString

The `OCECToRString` function converts a C string into an AOCE string.

```
pascal void OCECToRString (const char *cStr, CharacterSet charSet,
                           RString *rStr,
                           unsigned short rStrLength);
```

`cStr` A pointer to the C string you want to convert.

`charSet` The script code that the `OCECToRString` function uses for the `RString` structure's header.

`rStr` A pointer to an `RString` structure. You must allocate this.

`rStrLength`
The length, in bytes, of the `body` field of the `RString` structure, not including the length of the header information. If the C string is longer than the AOCE string, then only the number of bytes equal to the value of the `rStrLength` parameter are copied from the C string into the AOCE string.

DESCRIPTION

Given a C string and a `RString` structure that you supply, the `OCECToRString` function converts the C string into the `RString` structure. The `OCECToRString` function uses the `charSet` and `rStrLength` parameters to create the `RStringHeader` field of the new `RString` structure.

SPECIAL CONSIDERATIONS

If the C string is longer than the AOCE string, then only the number of bytes equal to the value of the `rStrLength` parameter are copied from the C string into the AOCE string.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0339

SEE ALSO

The `RString` structure is described on page 2-20.

For information on converting an `RString` structure to or from a Pascal string, see the functions `OCEPToRString` (next) and `OCERToPString` (page 2-48).

OCEPToRString

The `OCEPToRString` function converts a Pascal string into an AOCE string.

```
pascal void OCEPToRString(ConstStr255Param pStr,
                          CharacterSet charSet,
                          RString *rStr,
                          unsigned short rStrLength);
```

`pStr` A pointer to the Pascal string you want to convert.

`charSet` The script code that the `OCEPToRString` function uses for the `RString` structure's header.

`rStr` A pointer to an `RString` structure. You must allocate this.

`rStrLength`
The length, in bytes, of the `body` field of the `RString` structure, not including the length of the header information. If the Pascal string is longer than the AOCE string, then only the number of bytes equal to the value of the `rStrLength` parameter are copied from the Pascal string into the AOCE string.

DESCRIPTION

The `OCEPToRString` function converts a Pascal string into an `RString` structure. The `OCEPToRString` function uses the `charSet` and `rStrLength` parameters to create the `RStringHeader` field of the new `RString` structure.

SPECIAL CONSIDERATIONS

If the Pascal string is longer than the AOCE string, then only the number of bytes equal to the value of the `rStrLength` parameter are copied from the Pascal string into the AOCE string.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$033A

SEE ALSO

The `RString` structure is described on page 2-20.

For information on converting an `RString` structure to a Pascal string, see the function `OCERToPString`, described next.

OCERToPString

The `OCERToPString` function converts an `RString` structure into a Pascal string.

```
pascal StringPtr OCERToPString (const RString *rStr);
```

`rStr` A pointer to an `RString` structure that you want to convert into a Pascal string.

DESCRIPTION

The `OCERToPString` function converts an `RString` structure into a Pascal string. As with all of the AOCE utility functions, no memory is allocated by this function, so the string pointer that is returned points directly back into the `RString` structure that you supply when you make the call.

You must check the character set, or script code of the AOCE string before calling the `OCEPToRString` function to determine how to handle the Pascal string returned by this function. Because `RString` structures contain character set information and Pascal strings do not, you need to decide how to interpret the Pascal string that is returned, because it may contain multibyte characters.

SPECIAL CONSIDERATIONS

You should check the length of the AOCE string that the `rStr` parameter points to before calling this function to see if the string is greater than 255 bytes. Because a Pascal string can contain at most 255 bytes, the `OCERToPString` function truncates the length of the Pascal string to the lower byte of the length of the AOCE string.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$033B

SEE ALSO

The `RString` structure is described on page 2-20.

To convert a Pascal string to an `RString` structure, use the function `OCEPToRString` described on page 2-47.

OCERelRString

The `OCERelRString` function compares two `RString` structures to determine their relative sorting order.

```
pascal short OCERelRString (const void *str1, const void *str2,
                            RStringKind kind);
```

`str1`	A pointer to the first `RString` structure you want to compare.
`str2`	A pointer to the second `RString` structure you want to compare
`kind`	The value the `OCERelRString` function uses to determine the proper method of comparing the two `RString` structures. See the description of the `RStringKind` type on page 2-24 for a complete definition of the different values for the `kind` parameter, and for the restrictions on when to use them.

DESCRIPTION

Given two `RString` structures pointed to by the parameters `str1` and `str2`, the `OCERelRString` function determines if the first AOCE string is greater than, equal to, or less than the second AOCE string. The `OCERelRString` uses the value of the `kind` parameter to determine how to compare the two `RString` structures. For certain kinds of `RString` structures, this function uses the International Utilities to compare them. Because the Text Utilities take into account primary and secondary ordering, this call will not return the value `sortsEqual` if the strings differ only in case (“Dave” is not equal to “dave”). For more information see the chapter “Text Utilities” in *Inside Macintosh: Text*.

The `OCERelRString` function can return the following values:

`sortsBefore`	–1	The first `RString` structure should sort before the second `RString` structure
`sortsEqual`	0	The two `RString` structures are equal
`sortsAfter`	1	The first `RString` structure should sort after the second `RString` structure

The result returned by the `OCERelRstring` function is undefined when either the `str1` parameter or the `str2` parameter is set to `nil`.

SPECIAL CONSIDERATIONS

Although this function uses the Text Utilities for comparing certain kinds of `RString` structures, it is still alright to call this routine at interrupt level.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032D

SEE ALSO

The `RString` structure is described on page 2-20.

To compare two `RString` structures for equality only, use the `OCEEqualRString` function, described next.

OCEEqualRString

The `OCEEqualRString` function checks the equality of two `RString` structures.

```
pascal Boolean OCEEqualRString (const void *str1, const void
                                *str2, RStringKind kind);
```

`str1`	A pointer to the first `RString` structure you want to compare.
`str2`	A pointer to the second `RString` structure you want to compare.
`kind`	A value that defines what kind of `RString` structures the `OCEEqualRString` function is comparing.

DESCRIPTION

Given pointers to two `RString` structures, the `OCEEqualRString` function compares them for equality, and returns `true` if they are equal, `false` if they are not. If the two AOCE strings have the same length, then they are compared for equality, with the method of comparison dependent upon the value of the `kind` parameter. If the two AOCE strings have different lengths, then they are not equal. For certain kinds of `RString` structures, this function uses the Text Utilities to compare the strings. For more information see the chapter "Text Utilities" in *Inside Macintosh: Text*. See the description of the `RStringKind` type on page 2-24 for a complete definition of the different values for the `kind` parameter, and for the restrictions on when to use them.

SPECIAL CONSIDERATIONS

Although this function uses the Text Utilities for comparing certain kinds of `RString` structures, it is still alright to call this routine at interrupt level.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0316

SEE ALSO

The `RString` structure is described on page 2-20.

The `RStringKind` structure is described on page 2-24.

OCEValidRString

The `OCEValidRString` function checks the validity of an AOCE string.

```
pascal Boolean OCEValidRString (const void *str,RStringKind kind);
```

`str`	A pointer to the AOCE string you want to validate.
`kind`	The kind of AOCE string being validated.

DESCRIPTION

The `OCEValidRString` function checks the AOCE string you supply for validity based on the type of AOCE string specified by the `kind` parameter and returns `true` if the AOCE string structure is valid, `false` if it is not. See the description of the `RStringKind` type on page 2-24 for a complete definition of the different values for the `kind` parameter, and for the restrictions on when to use them. Currently this function checks for validity by ensuring that the length of the AOCE string is the proper size for its particular type. A `nil` pointer and a length of 0 for the `RString` structure are considered valid.

SPECIAL CONSIDERATIONS

The `OCEValidRString` function may be modified in the future to perform other checks for validity, so you should not assume that the only thing this function examines is the length of the AOCE string.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0338

SEE ALSO

The `RString` structure is described on page 2-20.

Creation Identifier Functions

The functions described in this section manipulate record and attribute creation identifiers in various ways. The two creation identifier data types are defined by the `CreationID` and `AttributeCreationID` structures, which are described on page 2-26.

OCEEqualCreationID

The `OCEEqualCreationID` function checks the equality of two `CreationID` structures.

```
pascal Boolean OCEEqualCreationID(const CreationID *cid1,
                                  const CreationID *cid2);
```

`cid1` A pointer to the first `CreationID` structure you want to compare.

`cid2` A pointer to the second `CreationID` structure you want to compare.

DESCRIPTION

Given pointers to two `CreationID` structures, `OCEEqualCreationID` compares the `CreationID` structures for equality, and returns `true` if their values are identical, `false` if they are not. Two `CreationID` structures are considered equal if each field in the first `CreationID` structure contains the same value as the corresponding field in the second `CreationID` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$030C

SEE ALSO

The `CreationID` structure is described on page 2-26.

OCECopyCreationID

The `OCECopyCreationID` function copies one `CreationID` structure to another.

```
pascal void OCECopyCreationID(const CreationID *cid1,
                              CreationID *const cid2);
```

`cid1` A pointer to the source `CreationID` structure you want to copy from. You must provide this structure.

`cid2` A pointer to the destination `CreationID` structure you want to copy to. You must provide this structure.

DESCRIPTION

Given two `CreationID` structures pointed to by the parameters, `cid1` and `cid2`, the `OCECopyCreationID` function copies the contents of the first structure to the second.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0300

SEE ALSO

The `CreationID` structure is described on page 2-26.

OCENullCID

The `OCENullCID` function returns a pointer to a null `CreationID` structure.

```
pascal const CreationID *OCENullCID(void);
```

DESCRIPTION

The `OCENullCID` function returns a pointer to a null `CreationID` structure that is maintained by the AOCE toolbox. You can use the `OCENullCID` function to check a `CreationID` structure to see if it is set to `NULL`, or to create a `NULL CID`. To check for a null `CreationID` structure you can use the following code fragment (This fragment uses the `OCEEqualCreationID` function described on page 2-52):

```
if (OCEEqualCreationID (myCID, OCENullCID())
/* then myCID is NULL */
```

You do not need to deallocate the `NULL CreationID` structure returned by the `OCENullCID` function when you are done with it.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0344

SEE ALSO

The `CreationID` structure is described on page 2-26.

To set an existing `CreationID` structure to `NULL`, call the `OCESetCreationIDtoNull` function (page 2-54).

The `OCECopyCreationID` function is described on page 2-52.

OCEPathFinderCID

The `OCEPathFinderCID` function returns a pointer to a special `CreationID` structure called the path finder creation ID.

```
pascal const CreationID *OCEPathFinderCID(void);
```

DESCRIPTION

The `OCEPathFinderCID` function returns a pointer to the special creation identifier structure known as the path finder creation ID. The path finder creation ID is maintained by the AOCE toolbox so you do not need to deallocate it when you are finished using it. This special creation ID is used by certain functions in the AOCE Authentication Manager. This function is intended for future use and is currently only used internally by the AOCE toolbox.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$033C

SEE ALSO

The `CreationID` structure is described on page 2-26.

OCESetCreationIDtoNull

The `OCESetCreationIDtoNull` function sets a `CreationID` structure to `NULL`.

```
pascal void OCESetCreationIDtoNull(CreationID *const cid);
```

`cid` A pointer to the `CreationID` structure you want to set to `NULL`.

DESCRIPTION

The `OCESetCreationIDtoNull` function sets the `CreationID` structure you provide to `NULL`. The `OCESetCreationIDtoNull` function makes it easier for you to use other AOCE functions such as `AuthResolveCreationID` that require the `CreationID` structure passed into them to be set to `NULL` before they are called.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032E

SEE ALSO

The `CreationID` structure is described on page 2-26.

For more information on the `AuthResolveCreationID` function see the chapter "Authentication Manager" in this book.

Packed Pathname Functions

The functions described in this section manipulate packed pathnames in various ways. The packed pathname is defined by the `PackedPathName` structure, which is described on page 2-29.

OCECopyPackedPathName

The `OCECopyPackedPathName` function copies the contents of one `PackedPathName` structure to another.

```
pascal OSErr OCECopyPackedPathName(const PackedPathName *path1,
                                   PackedPathName *path2,
                                   unsigned short path2Length);
```

`path1` A pointer to the source `PackedPathName` structure that you want to copy from.

`path2` A pointer to the destination `PackedPathName` structure that you want to copy to.

`path2Length`
The length, in bytes, of the `PackedPathName` structure pointed to by the `path2` parameter, not including the size information contained in the `dataLength` field.

DESCRIPTION

Given two `PackedPathName` structures pointed to by the parameters, `path1` and `path2`, the `OCECopyPackedPathName` function copies the contents of the first structure into the second. The `path2Length` parameter is the size, in bytes, of the destination `PackedPathName` structure excluding the size information contained in the `dataLength` field. The destination `PackedPathName` structure must be large enough to hold the entire contents of the source `PackedPathName` structure; otherwise, a memory-full error is returned. Therefore, when you allocate a new destination

PackedPathName structure as the destination, you must set its length field to the proper size before calling the OCECopyPackedPathName function.

You obtain the proper size for a PackedPathName structure from its dataLength field. Once you obtain this value, you can then use it to allocate a destination PackedPathName structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0304

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough memory to copy path1 into path2

SEE ALSO

The PackedPathName structure is described on page 2-29.

OCEIsNullPackedPathName

The OCEIsNullPackedPathName function determines if the value of a PackedPathName structure is NULL.

```
pascal Boolean OCEIsNullPackedPathName(const PackedPathName
                                       *path);
```

path A pointer to the PackedPathName structure you want to evaluate.

DESCRIPTION

Given a pointer to a PackedPathName structure, the OCEIsNullPackedPathName function determines if it satisfies the conditions for being considered NULL and returns true if its value is NULL, false if it is not. The value true is returned for any of the following conditions:

- If the path parameter is set to nil.
- If the PackedPathName structure pointed to by the path parameter has a length of 0.
- If the PackedPathName structure pointed to by the path parameter is composed of 0 RString components.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$031D

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

OCEPackedPathNameSize

The `OCEPackedPathNameSize` function computes the number of bytes required to create a `PackedPathName` structure, including the size information.

```
pascal unsigned short OCEPackedPathNameSize
                        (const RStringPtr parts[],
                         const unsigned short nParts);
```

`parts` An array of pointers to `RString` structures containing the dNode names.

`nParts` The number of individual dNode names that are contained in the `parts` array.

DESCRIPTION

The `OCEPackedPathNameSize` function computes the number of bytes of memory needed to hold a `PackedPathName` structure manufactured from the `parts` array. This length includes the size of the `dataLength` field of the `PackedPathName` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0328

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

For information on determining the number of partial pathnames within a `PackedPathName` structure see the `OCEDNodeNameCount` function, described next.

For information on packing and unpacking pathnames see the `OCEUnpackPathName` (page 2-58) and `OCEPackPathName` (page 2-60) functions.

OCEDNodeNameCount

The `OCEDNodeNameCount` function returns the number of `RString` structures, or catalog node names contained within a `PackedPathName` structure.

```
pascal unsigned short OCEDNodeNameCount
                                    (const PackedPathName *path);
```

`path` The `PackedPathName` structure that you want to evaluate.

DESCRIPTION

When you call the `OCEUnpackPathName` function to unpack a `PackedPathName` structure, you must pass it the number of dNodes that the path is composed of and allocate an array large enough to hold the pointers to each dNode name. The `OCEDNodeNameCount` function provides you with the number of dNodes contained in the path.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032C

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

For information on determining the size of a `PackedPathName` structure needed to hold all the components of a pathname, see the `OCEPackedPathNameSize` function on page 2-57.

For information on packing and unpacking pathnames see the `OCEUnpackPathName` (next) and `OCEPackPathName` (page 2-60) functions.

OCEUnpackPathName

The `OCEUnpackPathName` function unpacks a `PackedPathName` structure into its component `RString` structures.

```
pascal unsigned short OCEUnpackPathName(const PackedPathName
                                *path, RString *const parts[],
                                const unsigned short nParts);
```

`path` A pointer to the `PackedPathName` structure that you want unpacked.

`parts`	An array of pointers to `RString` structures that the `OCEUnpackPathName` function fills with pointers into the `path` parameter.
`nParts`	The size of the `parts` array.

DESCRIPTION

Given a pointer into a `PackedPathName` structure that you provide, the `OCEUnpackPathName` function breaks apart the structure specified by `path` into the individual `RString` structures it contains, writing pointers to these `RString` structures into the `parts` array. The `parts` array must be large enough to hold as many as `nParts` dNode names. You can determine the number of dNodes that a path contains by calling the `OCEDNodeNameCount` function (page 2-58).

The `OCEUnpackPathName` function returns the number of dNode names actually found during the process of unpacking. You should check this value to ensure that it corresponds to the `nParts` parameter that you supplied to verify that no discrepancies exist.

The `RString` structures are placed in the `parts` array in order from lowest to highest; that is, the first element beneath the top level in the `PackedPathName` structure is placed last in the `parts` array.

SPECIAL CONSIDERATIONS

The array in the `parts` parameter generated by the `OCEUnpackPathName` function contains pointers into the `PackedPathName` structure specified by the `path` parameter. You should not delete or reuse the `PackedPathName` structure pointed to by the `path` parameter until you are finished with the `parts` array as well. Otherwise, the parts array may no longer contain pointers to valid data.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0330

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

For information on packing a pathname structure, see the `OCEPackPathName` function, described next.

For information on determining the size of a `PackedPathName` structure needed to hold all the components of a pathname, see the `OCEPackedPathNameSize` function on page 2-57.

For information on determining the number of partial pathnames within a `PackedPathName` structure, see the `OCEDNodeNameCount` function described on page 2-58.

OCEPackPathName

The `OCEPackPathName` function forms a `PackedPathName` structure from its component parts.

```
pascal OSErr OCEPackPathName(const RStringPtr parts[],
                             const unsigned short nParts,
                             PackedPathName *path,
                             unsigned short pathLength);
```

`parts` An array of `RString` structures that the `OCEPackPathName` function uses to form the packed pathname.

`nParts` The number of dNodes contained in the `parts` array.

`path` A pointer to a buffer that you have allocated to hold the `PackedPathName` structure.

`pathLength`
The size of the structure pointed to by the `path` parameter, not including the size information contained in the `dataLength` field. For information on determining the size of a `PackedPathName` structure needed to hold all the components of a pathname, see the `OCEPackedPathNameSize` function on page 2-57.

DESCRIPTION

The `OCEPackPathName` function takes a buffer that you supply and stores in it a `PackedPathName` structure that the function creates from an array of dNodes. The buffer must be large enough to hold the full packed pathname. If the buffer is too small to hold the entire packed pathname, then a memory-full error is returned.

The order in which you store the partial pathnames in the `parts` array is as follows: `parts[0]` should contain the last pathname element, and `parts[nParts - 1]` should contain the name of the first pathname element beneath the root.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0323

RESULT CODES

noErr	0	No error
memFullErr	–108	The buffer pointed to by the path parameter is not large enough to hold the entire contents of the parts array.

SEE ALSO

The PackedPathName structure is described on page 2-29.

For information on unpacking a pathname structure, see the OCEUnpackPathName function described on page 2-58.

For information on determining the size of a PackedPathName structure needed to hold all the components of a pathname, see the OCEPackedPathNameSize function on page 2-57.

For information on determining the number of partial pathnames within a PackedPathName structure, see the OCEDNodeNameCount function described on page 2-58.

OCEEqualPackedPathName

The OCEEqualPackedPathName checks the equality of two packed pathnames.

```
pascal Boolean OCEEqualPackedPathName(const PackedPathName *path1,
                                      const PackedPathName *path2);
```

path1	A pointer to the first PackedPathName you want to compare.
path2	A pointer to the second PackedPathName you want to compare.

DESCRIPTION

Given pointers to two PackedPathName structures, path1 and path2, the OCEEqualPackedPathName function compares them for equality and returns true if the two pathnames are equal and false if they are not. This function takes into account the proper case and diacritical marks of the various fields of the PackedPathName structures it compares. This function checks for equality in the following manner:

- If the value of both PackedPathName structures is NULL, they are equal. A PackedPathName structure is considered NULL if the pointer to it is set to nil, or if its length is 0, or if it contains 0 catalog node names.
- If the value of one PackedPathName structure is NULL, but the value of the other is not, they are not equal.
- If neither PackedPathName structures is NULL, but they do not contain the same number of catalog node names, they are not equal.
- If neither PackedPathName structures is NULL and they both contain the same number of catalog node names, then each catalog node name is compared with the

corresponding one with regard to case and diacritical marks. If every one compares exactly, then the `PackedPathName` structures are equal. Otherwise, they are not.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0311

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

OCEValidPackedPathName

The `OCEValidPackedPathName` function checks a given `PackedPathName` structure for internal consistency.

```
pascal Boolean OCEValidPackedPathName(const PackedPathName *path);
```

`path` A pointer to the `PackedPathName` you want to validate.

DESCRIPTION

The `OCEValidPackedPathName` function returns `true` if the `PackedPathName` structure is valid; otherwise, it returns `false`. The `OCEValidPackedPathName` function checks the `PackedPathName` structure for validity by unpacking it and performing the following tests:

- If the pointer to the `PackedPathName` structure is set to `nil`, the `OCEValidPackedPathName` function considers the `PackedPathName` structure to be invalid and returns `false`.
- If the length of the `PackedPathName` structure is 0 it is considered valid.
- It checks that all of the catalog node names in the `PackedPathName` structure are valid by passing them to the `OCEValidRString` function (page 2-51).
- It adds up the lengths of all the catalog node names in the `PackedPathName` structure and verifies that the total length matches the length of the `PackedPathName` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0334

SEE ALSO

The `PackedPathName` structure is described on page 2-29.

Catalog Discriminator Functions

The utility functions described in this section manipulate catalog discriminators. The catalog discriminator is defined by the `DirDiscriminator` structure described on page 2-30.

OCECopyDirDiscriminator

The `OCECopyDirDiscriminator` function copies the value of one `DirDiscriminator` structure to another.

```
pascal void OCECopyDirDiscriminator
                    (const DirDiscriminator *disc1,
                     DirDiscriminator *const disc2);
```

`disc1` A pointer to the source `DirDiscriminator` structure that you want to copy from. You must provide this structure.

`disc2` A pointer to the destination `DirDiscriminator` structure that you want to copy to. You must provide this structure.

DESCRIPTION

Given two `DirDiscriminator` structures pointed to by the parameters, `disc1` and `disc2`, the `OCECopyDirDiscriminator` function copies the contents of the first structure to the second.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0301

SEE ALSO

The `DirDiscriminator` structure is described on page 2-30.

OCEEqualDirDiscriminator

The `OCEEqualDirDiscriminator` function checks the equality of two `DirDiscriminator` structures.

```
pascal Boolean OCEEqualDirDiscriminator
                    (const DirDiscriminator *disc1,
                     DirDiscriminator *const disc2);
```

`disc1` A pointer to the first `DirDiscriminator` structure you want to compare.

`disc2` A pointer to the second `DirDiscriminator` structure you want to compare.

DESCRIPTION

Given pointers to two `DirDiscriminator` structures, the `OCEEqualDirDiscriminator` function determines if they are equal. It returns `true` if they are equal, `false` if they are not. The two `DirDiscriminator` structures are considered equal if their `signature` and `misc` fields match byte for byte.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$030D

SEE ALSO

The `DirDiscriminator` structure is described on page 2-30.

Record Location Information Functions

The functions described in this section manipulate record location information structures. The record location information structure is defined by the `RLI` data type, described on page 2-32.

OCENewRLI

The `OCENewRLI` function constructs an `RLI` structure from its component parts.

```
pascal void OCENewRLI(RLI *newRLI, const DirectoryName *dirName,
                                DirDiscriminator *discriminator,
                                const DNodeNum dNodeNumber,
                                const PackedPathName *path);
```

`newRLI` A pointer to the buffer where the `OCENewRLI` function stores the `RLI` structure it constructs. You must allocate this.

`dirName` A pointer to the catalog name you want incorporated into the `RLI` structure.

`discriminator` A pointer to the catalog discriminator you want incorporated into the `RLI` structure.

`dNodeNumber` The catalog node number you want incorporated into the `RLI` structure.

`path` A pointer to the packed pathname you want incorporated into the `RLI` structure.

DESCRIPTION

Given catalog name, discriminator, catalog node number, and packed pathname structures, the `OCENewRLI` function creates an `RLI` structure and replaces the contents of the buffer, `newRLI`, with the `RLI` structure that it forms.

SPECIAL CONSIDERATIONS

Because the `OCENewRLI` function does not allocate any memory, the `RLI` structure it forms uses the same `DirectoryName` structure and the same `PackedPathname` structure that you supplied as parameters. Therefore, you should not dispose of or reuse the `DirectoryName` and `PackedPathname` structures until you have finished using the `RLI` structure as well. Doing so will cause the pointers in the `RLI` structure to point to incorrect locations in memory and might cause your application to crash.

Use `OCENewRLI` instead of the `OCEUnPackRLI` function to create an `RLI` structure that you are going to make an alias for. An alias to an `RLI` structure created with the `OCEUnPackRLI` function does not work properly.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$031F

SEE ALSO

The `RLI` structure is described on page 2-32.

The `DirDiscriminator` structure is described on page 2-30.

OCEDuplicateRLI

The `OCEDuplicateRLI` function duplicates the contents of one `RLI` structure to another.

```
pascal void OCEDuplicateRLI(const RLI *rli1, RLI *rli2);
```

`rli1` A pointer to the source `RLI` structure. You must allocate this structure.

`rli2` A pointer to the destination `RLI` structure. You must allocate this structure; however, you do not have to allocate the structures that this `RLI` structure points to.

DESCRIPTION

The `OCEDuplicateRLI` function copies the pointers from the `directoryName` and `path` fields of the source RLI structure to the corresponding fields in the destination `RLI` structure. This function does not copy the data these fields point to, only the pointers to the data. After you call the `OCEDuplicateRLI` function, each `RLI` structure contains pointers to the same `PackedPathName` and `DirectoryName` structures. This means that if you free the memory for one `RLI` structure's `PackedPathName` or `DirectoryName` structure, you are freeing the same structure in the corresponding `RLI` structure as well. In addition, the `OCEDuplicateRLI` function copies the values from the source `RLI` structure's `dirDiscriminator` and `dNodeNumber` fields into the corresponding fields of the destination `RLI` structure.

To actually copy the contents of the structures that the `DirectoryNamePtr` and `PackedPathNamePtr` fields point to from one `RLI` to another, use the `OCECopyRLI` function, described next.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$030B

SEE ALSO

The `RLI` structure is described on page 2-32.

To copy the contents of one `RLI` structure to another see the `OCECopyRLI` function, described next.

To copy one `PackedRLI` structure to another see the `OCECopyPackedRLI` function on page 2-70.

For a description of the difference between copying and duplicating an `RLI` structure, see the section "Copying Versus Duplicating AOCE Data Structures" on page 2-15.

OCECopyRLI

The `OCECopyRLI` function copies the contents of one `RLI` structure into another.

```
pascal OSErr OCECopyRLI(const RLI *rli1, RLI *rli2);
```

`rli1`	A pointer to the source `RLI` structure. You must allocate this structure.
`rli2`	A pointer to the destination `RLI` structure. You must allocate this structure.

DESCRIPTION

Given pointers to two `RLI` structures pointed to by the parameters, `rli1` and `rli2`, the `OCECopyRLI` function copies the contents of the first into the second. The destination `RLI` structure must already contain pointers to structures large enough to hold copies of the corresponding fields from the source `RLI` structure; otherwise, a memory-full error is returned. Therefore, when you allocate a new destination `RLI` structure, you must set the fields that define the length of the `PackedPathName` and `DirectoryName` structures pointed to by its `path` and `directoryName` fields to the proper size before calling the `OCECopyRLI` function.

You obtain the proper size for a `PackedPathName` from its `dataLength` field and that of a `DirectoryName` structure from its `RStringHeader`. Once you obtain these values, you can then use them to allocate structures of the correct size.

If you want a destination `RLI` structure that points to the same `PackedPathName` and `DirectoryName` structures as the source `RLI` structure, then use the `OCEDuplicateRLI` function (page 2-66). The `OCEDuplicateRLI` function changes the destination `RLI` structure's `path` and `directoryName` fields so that they point to the same data in the fields of the corresponding source `RLI` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0307

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The destination `RLI` structure is not large enough to hold the entire contents of the source `RLI` structure.

SEE ALSO

The `RLI` structure is described on page 2-32.

To create a destination RLI structure that points to the same PackedPathName and DirectoryName structures as the source RLI structure, use the OCEDuplicateRLI function on page 2-66.

To copy one PackedRLI structure to another see the OCECopyPackedRLI function on page 2-70.

The PackedPathName and DirectoryName structures are described on page 2-29 and page 2-22, respectively.

For a description of the difference between copying and duplicating an RLI structure, see the section "Copying Versus Duplicating AOCE Data Structures" on page 2-15.

OCEEqualRLI

The OCEEqualRLI function checks the equality of two record location information structures.

```
pascal Boolean OCEEqualRLI(const RLI *rli1, const RLI *rli2);
```

rli1	A pointer to the first RLI structure you want to compare.
rli2	A pointer to the second RLI structure you want to compare.

DESCRIPTION

Given pointers to two RLI structures, the OCEEqualRLI function compares them for equality and returns true if they are equal, false if they are not. This function takes into account differences in the case and diacritical marks of the catalog name and the pathname that are contained in the RLI structures.

If the RLI structure that the rli1 parameter points to contains a catalog node number and a nil pathname, and the RLI structure that the rli2 parameter points to contains the value kNULLDNodeNumber and a pathname that is not nil, then the comparison will fail. In other words, the two RLI structures must be of the same form before they can be compared for equality. The one exception to this rule is when the pathname contained in the two RLI structures is set to nil. In that case, a dNodeNumber field with a value of kNULLDNodeNumber, and a dNodeNumber field with a value of kRootDNodeNumber are treated as equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0315

SEE ALSO

The RLI structure is described on page 2-32.

To check two PackedRLI structures for equality, use the OCEEqualPackedRLI function (page 2-76).

OCEValidRLI

The OCEValidRLI function checks the validity of a record location information structure.

```
pascal Boolean OCEValidRLI(const RLI *theRLI);
```

theRLI A pointer to the RLI structure you want to check.

DESCRIPTION

The OCEValidRLI function returns true if the RLI structure is valid, false if it is not. It checks for validity in the following manner:

- If the pointer to the RLI structure is set to nil, then the OCEValidRLI function considers the RLI structure to be invalid and returns false.
- The OCEValidRLI function then checks if the catalog name length is greater than 0 and less than or equal to the constant kDirectoryNameMaxBytes. If it is not, then the RLI structure is not valid.
- The OCEValidRLI function then checks that the packed pathname, if specified, is valid by calling the OCEValidPackedPathName function (page 2-62). If the OCEValidPackedPathName function returns false, the RLI structure is not valid.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0337

SEE ALSO

The RLI structure is described on page 2-32.

To perform a validity check on a PackedRLI structure use the OCEValidPackedRLI function (page 2-77).

OCECopyPackedRLI

The `OCECopyPackedRLI` function copies the contents of one `PackedRLI` structure into another.

```
pascal OSErr OCECopyPackedRLI(const PackedRLI *prli1,
                              PackedRLI *prli2,
                              unsigned short prli2Length);
```

`prli1` A pointer to the source `PackedRLI` structure.

`prli2` A pointer to the destination `PackedRLI` structure.

`prli2Length`
The size of the destination `PackedRLI` structure, not including the size of the `dataLength` field.

DESCRIPTION

Given two `PackedRLI` structures pointed to by the parameters `prli1` and `prli2`, the `OCECopyPackedRLI` function copies the contents of the first `PackedRLI` structure into the second. The `prli2Length` parameter is the size of the destination `PackedRLI` structure, excluding its `dataLength` field. The destination `PackedRLI` structure must be large enough to hold the entire contents of the source `PackedRLI` structure; otherwise, a memory-full error is returned.

You obtain the proper size for a `PackedRLI` structure from its `dataLength` field. Once you obtain this value, you can then use it to allocate a `PackedRLI` structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0305

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The destination `PackedRLI` structure is not large enough to hold the contents of the source `PackedRLI` structure

SEE ALSO

The `PackedRLI` structure is described on page 2-33.

To copy an `RLI` structure, use the `OCECopyRLI` function (page 2-67).

OCEPackedRLISize

The `OCEPackedRLISize` function computes the number of bytes of memory needed to hold a `PackedRLI` structure.

```
pascal unsigned short OCEPackedRLISize(const RLI *theRLI);
```

`theRLI` A pointer to an `RLI` structure.

DESCRIPTION

Given a pointer to an `RLI` structure, the `OCEPackedRLISize` function computes the number of bytes needed for a `PackedRLI` structure large enough to hold the data in the `RLI` structure. The number of bytes returned by the `OCEPackedRLISize` function includes the bytes in the field that specifies the length of the `PackedRLI` structure, which enables you to allocate the correct amount of memory for a `PackedRLI` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032A

SEE ALSO

The `RLI` structure is defined on page 2-32.

The `PackedRLI` structure is defined on page 2-33.

To obtain the number of bytes necessary to create a `PackedRLI` structure from the component parts of an `RLI` structure, see the `OCEPackedRLIPartsSize` function on page 2-73.

OCEPackRLI

The `OCEPackRLI` function packs a record location information structure.

```
pascal OSErr OCEPackRLI(const RLI *theRLI, PackedRLI *prli,
                        unsigned short prliLength);
```

`theRLI` A pointer to the record location information structure you want packed.

`prli` A pointer to a `PackedRLI` structure. You must allocate this.

`prliLength`
The length, in bytes, of the `PackedRLI` structure pointed to by the `prli` parameter, excluding the bytes in the `dataLength` field.

DESCRIPTION

The `OCEPackRLI` function packs the contents of an `RLI` structure into a `PackedRLI` structure. During this process, the `OCEPackRLI` function replaces the contents of the `PackedRLI` structure with new data from the `RLI` structure. The `PackedRLI` structure must be large enough to hold the contents of the `RLI` structure when packed; otherwise, a memory-full error is returned. To determine the correct size for the `PackedRLI` structure, call the `OCEPackedRLISize` function (page 2-71).

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0324

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The `PackedRLI` structure is not large enough to hold the contents of the `RLI` structure when packed

SEE ALSO

The `RLI` structure is defined on page 2-32.

The `PackedRLI` structure is defined on page 2-33.

For information on unpacking a `PackedRLI` structure see the `OCEUnpackRLI` function, next.

To create a `PackedRLI` structure from the component parts of an `RLI` structure, use the `OCEPackRLIParts` function on page 2-74.

To determine the correct size for the `PackedRLI` structure, call the `OCEPackedRLISize` function on page 2-71.

OCEUnpackRLI

The `OCEUnpackRLI` function unpacks a `PackedRLI` structure into its component parts.

```
pascal void OCEUnpackRLI(const PackedRLI *prli, RLI *theRLI);
```

`prli1` A pointer to the `PackedRLI` structure you want unpacked.

`theRLI` A pointer to the `RLI` structure. You must allocate this; however, you do not have to allocate the structures that this `RLI` structure points to.

DESCRIPTION

Given a `PackedRLI` structure pointed to by the `prli1` parameter, and an `RLI` structure pointed to by the parameter `theRLI`, the `OCEUnpackRLI` function unpacks the `PackedRLI` structure into its components, writing pointers to these components into the `RLI` structure that you supply.

SPECIAL CONSIDERATIONS

The unpacked `RLI` structure contains pointers into the packed structure. Therefore, you should not delete or reuse the packed structure pointed to by the `prli1` parameter until you are finished with the unpacked `RLI` structure as well.

An alias to an `RLI` structure created with the `OCEUnPackRLI` function does not work properly. If you unpack an `RLI` structure with `OCEUnPackRLI`, create an alias to it, and then pack it with `OCEPackRLI`, when you try to extract the alias with `OCEExtractAlias`, a `nil` value is returned for the new `PackedRLI` structure. Use the `OCENewRLI` function (page 2-64) instead of `OCEUnPackRLI` whenever you create an `RLI` structure with an alias.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0331

SEE ALSO

The `RLI` structure is defined on page 2-32.

The `PackedRLI` structure is defined on page 2-33.

For information on packing an `RLI` structure see the `OCEPackRLI` function on page 2-71.

OCEPackedRLIPartsSize

The `OCEPackedRLIPartsSize` function computes the size of a `PackedRLI` structure needed to hold the constituent parts of an `RLI` structure.

```
pascal unsigned short OCEPackedRLIPartsSize
                        (const DirectoryName *dirName,
                         const RStringPtr parts[],
                         const unsigned short nParts);
```

`dirName`	A pointer to a catalog name structure.
`parts`	An array containing the pathname parts.
`nParts`	The number of parts contained in the `parts` array.

DESCRIPTION

Given the component parts of a record location information structure, the `OCEPackedRLIPartsSize` function returns the size, in bytes, needed to create a `PackedRLI` structure large enough to hold all of the data and the `PackedRLI` `dataLength` field. This function is equivalent to the `OCEPackedRLISize` function (page 2-71), except that it takes the parts of an `RLI` structure as parameters instead of an `RLI` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0329

SEE ALSO

The `RLI` structure is defined on page 2-32.

The `PackedRLI` structure is defined on page 2-33.

To pack the components of an `RLI` structure into a `PackedRLI` structure, see the `OCEPackRLIParts` function, described next.

OCEPackRLIParts

The `OCEPackRLIParts` function packs the components of a record location information structure into a `PackedRLI` structure.

```
pascal OSErr OCEPackRLIParts(const DirectoryName *dirName,
                             const DirDiscriminator *discriminator,
                             const DNodeNum dNodeNumber,
                             const RStringPtr parts[],
                             const unsigned short nParts,
                             PackedRLI *prli,
                             unsigned short prliLength);
```

`dirName` A pointer to a catalog name structure you want packed.

`discriminator`
A pointer to a `DirDiscriminator` value you want packed.

`dNodeNumber`
The catalog node number you want packed.

`parts` An array of pointers to `RString` structures, each of which is a dNode name on the path. The total array is the pathname structure that you want packed.

`nParts` The number of dNode names contained in the `parts` array.

`prli`	A pointer to a `PackedRLI` structure that you have allocated.
`prliLength`	The length, in bytes, of the `PackedRLI` structure pointed to by the `prli` parameter.

DESCRIPTION

From all of the component pieces of a record location information structure, the `OCEPackRLIParts` function forms a `PackedRLI` structure. You must allocate the storage for the `PackedRLI` structure before calling this function. This function is equivalent to the `OCEPackRLI` function, except that it takes the parts of an `RLI` structure as its parameters instead of an `RLI` structure. The `OCEPackRLIParts` function examines the `prliLength` parameter to see if the structure pointed to by the `prli` parameter is large enough to hold the packed contents of the `RLI` structure, and returns a memory-full error if it is not. Use the `OCEPackedRLIPartsSize` function to obtain the size needed for a `PackedRLI` structure large enough to hold the data from all of the pieces of an `RLI` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0325

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The `PackedRLI` structure is not large enough to hold the packed components of the `RLI` structure

SEE ALSO

The `RLI` structure is defined on page 2-32.

The `PackedRLI` structure is defined on page 2-33.

For information on unpacking a `PackedRLI` structure see the `OCEUnpackRLI` function on page 2-72.

To obtain the number of bytes necessary to create a `PackedRLI` structure from the component parts of an `RLI` structure, see the `OCEPackedRLIPartsSize` function on page 2-73.

OCEEqualPackedRLI

The `OCEEqualPackedRLI` function checks the equality of two packed record location information structures.

```
pascal Boolean OCEEqualPackedRLI(const PackedRLI *prli1,
                                 const PackedRLI *prli2);
```

`prli1` A pointer to the first `PackedRLI` structure you want to compare.

`prli2` A pointer to the second `PackedRLI` structure you want to compare.

DESCRIPTION

The `OCEEqualPackedRLI` function determines if two `PackedRLI` structures are equal and returns `true` if they are, `false` if they are not. This function checks for equality in the following manner:

- If the value of both `PackedRLI` structures is `NULL` they are equal. The `PackedRLI` structures are set to `NULL` if the pointers to them are `nil`, or if they have a length of 0.
- If only one `PackedRLI` structure is `NULL`, the `PackedRLI` structures are not equal.
- If neither `PackedRLI` structures is `NULL`, then they are unpacked and their `discriminator` and `dNodeNumber` field's values are compared. If these values are not identical, then the `PackedRLI` structures are not equal. If the values are identical, then the `DirectoryName` and `PackedPathName` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) and `OCEEqualPackedPathName` (page 2-61) functions. If the `DirectoryName` and `PackedPathName` structures are equal then the `PackedRLI` structures are equal; otherwise, the `PackedRLI` structures are not equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0313

SEE ALSO

The `PackedRLI` structure is defined on page 2-33.

To check the equality of two `RLI` structures use the `OCEEqualRLI` function (page 2-68).

OCEValidPackedRLI

The `OCEValidPackedRLI` function checks the validity of a packed record location information structure.

```
pascal Boolean OCEValidPackedRLI(const PackedRLI *prli);
```

`prli` A pointer to a `PackedRLI` structure.

DESCRIPTION

The `OCEValidPackedRLI` function checks a `PackedRLI` structure for validity and returns `true` if it is valid, `false` if it is not. The `OCEValidPackedRLI` function determines validity by unpacking the `PackedRLI` structure and then performing the following tests on it:

- If the pointer to the `PackedRLI` structure is `nil`, or the `PackedRLI` structure has a length of 0, then the `PackedRLI` structure is not valid.
- The `OCEValidPackedRLI` function determines if the `PackedRLI` structure is larger than the smallest possible `PackedRLI` structure. If it is not, then the `PackedRLI` structure is not valid.
- The `OCEValidPackedRLI` function then checks that the catalog name of the `PackedRLI` structure is valid by calling the `OCEValidRString` function (page 2-51). If the `OCEValidRString` function returns `false`, then the `PackedRLI` structure is not valid.
- The `OCEValidPackedRLI` function then checks the validity of the packed pathname of the `PackedRLI` structure by calling the `OCEValidPackedPathName` function (page 2-62). If the `OCEValidPackedPathName` function returns `false`, then the `PackedRLI` structure is not valid.
- The `OCEValidPackedRLI` function then adds up the sizes of all of the fields in the `PackedRLI` structure and compares the total number of bytes to the value contained in the `dataLength` field of the `PackedRLI` structure. If the two values are equal, then the `PackedRLI` structure is valid; otherwise, it is not valid.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0336

SEE ALSO

The `PackedRLI` structure is defined on page 2-33.

To check the validity of an `RLI` structure, use the `OCEValidRLI` function (page 2-69).

OCEExtractAlias

The `OCEExtractAlias` function returns an alias record from a packed record location information structure.

```
pascal AliasPtr OCEExtractAlias(const PackedRLI *prli);
```

`prli` A pointer to the `PackedRLI` structure containing the alias you want to extract.

DESCRIPTION

If the `PackedRLI` structure describes a personal catalog, the `OCEExtractAlias` function extracts an HFS alias to the personal catalog.

To use the alias, connect it to a handle and call the `ResolveAlias` function as shown in the following code sample.

```
aliasPtr = OCEExtractAlias()
status = PtrToHand(
   (Ptr) aliasPtr,
   (Handle *) &aliasHandle,
   aliasPtr->aliasSize
   );
if (status == noErr)
status = ResolveAlias(NULL, aliasHandle, &theFSSpec, &wasChanged);
```

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0318

SEE ALSO

The `PackedRLI` structure is defined on page 2-33.

See the chapter "Alias Manager" in *Inside Macintosh: Files* for more information on aliases and the alias structure.

OCEGetDirectoryRootPackedRLI

The `OCEGetDirectoryRootPackedRLI` function returns a pointer to a special packed `RLI` structure that represents the root of all catalogs.

```
pascal const PackedRLI * OCEGetDirectoryRootPackedRLI (void)
```

DESCRIPTION

You use the `OCEGetDirectoryRootPackedRLI` function whenever you need to obtain the record location information for the root of all catalogs. This `PackedRLI` structure is maintained by the AOCE toolbox, and therefore you never need to free the `PackedRLI` structure returned by the `OCEGetDirectoryRootPackedRLI` function when you have finished using it.

Clients of the AOCE standard catalog-browsing panel can use the `PackedRLI` returned by this function to tell the Standard Catalog panel to begin displaying catalogs from the root, thus allowing the user to see all of the catalogs configured on the computer.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0346

SEE ALSO

The `PackedRLI` structure is defined on page 2-33.

The catalog-browsing panel is described in the chapter "Standard Catalog Package" in this book.

Local Record Identifier Functions

The functions described in this section manipulate local record identifier structures. The local record identifier is defined by the `LocalRecordID` structure (page 2-27).

OCENewLocalRecordID

The `OCENewLocalRecordID` function converts the data you supply into a `LocalRecordID` structure.

```
pascal void OCENewLocalRecordID(const RString *recordName,
                                const RString *recordType,
                                const CreationID *cid,
                                LocalRecordID *lRID);
```

`recordName`
A pointer to an `RString` structure containing the record name you want stored in the `LocalRecordID` structure.

`recordType`
A pointer to an `RString` structure containing the record type you want stored in the `LocalRecordID` structure.

cid	A pointer to the CreationID structure you want stored in the LocalRecordID structure.
lRID	A pointer to a LocalRecordID structure you have allocated.

DESCRIPTION

The OCENewLocalRecordID function converts a record name, record type, and creation identifier into a LocalRecordID structure. You must allocate the storage for the LocalRecordID structure before calling this function.

SPECIAL CONSIDERATIONS

Because the OCENewLocalRecordID function does not allocate any memory, the LocalRecordID structure it forms uses the same RString structures and the same CreationID structure that you supplied as parameters. Therefore, you should not dispose of or reuse the RSTring and CreationID structures until you have finished using the LocalRecordID structure as well. Doing so will cause the pointers in the LocalRecordID structure to point to incorrect locations in memory and might cause your application to crash.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$031E

SEE ALSO

The LocalRecordID structure is defined on page 2-27.

OCECopyLocalRecordID

The OCECopyLocalRecordID function copies one LocalRecordID structure into another.

```
pascal OSErr OCECopyLocalRecordID(const LocalRecordID *lRID1,
                                  LocalRecordID *lRID2);
```

lRID1	A pointer to the source LocalRecordID structure.
lRID2	A pointer to the destination LocalRecordID structure.

DESCRIPTION

Given two LocalRecordID structures, the OCECopyLocalRecordID function copies the contents of the first one into the second. The destination LocalRecordID structure

must contain pointers to `RString` structures large enough to hold copies of the corresponding fields from the source `LocalRecordID` structure; otherwise, a memory-full error is returned. Therefore, when you allocate a new destination `LocalRecordID` structure, you must set the length fields of the `RString` structures pointed to by `recordName` and `recordType` to their proper values before calling the `OCECopyLocalRecordID` function. You obtain the correct size for these `Rstring` structures from their headers in the source `LocalRecordID` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0302

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The destination `LocalRecordID` structure is not large enough to hold the contents of the source `LocalRecordID` structure

SEE ALSO

The `LocalRecordID` structure is defined on page 2-27.

OCEEqualLocalRecordID

The `OCEEqualLocalRecordID` function checks the equality of two `LocalRecordID` structures.

```
pascal Boolean OCEEqualLocalRecordID(const LocalRecordID *lRID1,
                                     const LocalRecordID *lRID2);
```

`lRID1` A pointer to the first `LocalRecordID` structure you want to compare.

`lRID2` A pointer to the second `LocalRecordID` structure you want to compare.

DESCRIPTION

The `OCEEqualLocalRecordID` function compares the two `LocalRecordID` structures for equality in the following manner:

- The `recordName` and `recordType` fields of the two `LocalRecordID` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) function and passing it the proper `RStringKind` value for each field.
- The `cid` fields of the two `LocalRecordID` structures are compared for equality by calling the `OCEEqualCreationID` function (page 2-52).

If the `recordName`, `recordType`, and `CreationID` fields of the two `LocalRecordID` structures are equal, then the `OCEEqualLocalRecordID` function returns `true`; otherwise, it returns `false`.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$030E

SEE ALSO

The `LocalRecordID` structure is defined on page 2-27.

The `RStringKind` structure is defined on page 2-24.

Short Record Identifier Functions

The functions described in this section manipulate short record identifiers. The short record identifier is defined by the `ShortRecordID` structure (page 2-35).

OCENewShortRecordID

The `OCENewShortRecordID` function converts data you supply into a `ShortRecordID` structure.

```
pascal void OCENewShortRecordID(const PackedRLI *theRLI,
                                const CreationID *cid,
                                ShortRecordIDPtr *sRID);
```

`theRLI`	A pointer to the packed record location information structure containing data you want stored in the `ShortRecordID` structure.
`cid`	A pointer to the creation identifier structure containing data you want stored in the `ShortRecordID` structure.
`sRID`	A pointer to a `ShortRecordID` structure you have allocated.

DESCRIPTION

The `OCENewShortRecordID` function converts a `CreationID` structure and a `PackedRLI` structure into a `ShortRecordID` structure. You must allocate the `ShortRecordID` structure before calling this function.

SPECIAL CONSIDERATIONS

Because the `OCENewRecordID` function does not allocate any memory, the `ShortRecordID` structure it forms uses the same `PackedRLI` structure and the same

CreationID structure that you supplied as parameters. Therefore, you should not dispose of or reuse the PackedRLI and CreationID structures until you have finished using the ShortRecordID structure as well. Doing so will cause the pointers in the ShortRecordID structure to point to incorrect locations in memory and might cause your application to crash.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0321

SEE ALSO

The ShortRecordID structure is defined on page 2-35.

OCECopyShortRecordID

The OCECopyShortRecordID function copies one ShortRecordID structure into another.

```
pascal OSErr OCECopyShortRecordID(const ShortRecordID *sRID1,
                                  ShortRecordID *sRID2);
```

sRID1	A pointer to the source ShortRecordID structure.
sRID2	A pointer to the destination ShortRecordID structure.

DESCRIPTION

Given two ShortRecordID structures pointed to by the sRID1 and sRID2 parameters, the OCECopyShortRecordID function copies the data from the first one into the second. The destination ShortRecordID structure must contain pointers to structures large enough to hold copies of the corresponding fields from the source ShortRecordID structure; otherwise, a memory-full error is returned. Therefore, when you allocate a new destination ShortRecordID structure, you must set the dataLength field of its PackedRLI component to the proper value before calling the OCECopyShortRecordID function.

You obtain the correct size for a PackedRLI structure from the value contained in its dataLength field. Once you obtain this value, you can then use it to allocate a PackedRLI structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$030A

RESULT CODES

noErr	0	No error
memFullErr	–108	The destination ShortRecordID structure is not large enough to hold the contents of the source ShortRecordID structure

SEE ALSO

The ShortRecordID structure is defined on page 2-35.

OCEEqualShortRecordID

The OCEEqualShortRecordID function checks the equality of two short record identifier structures.

```
pascal Boolean OCEEqualShortRecordID(const ShortRecordID *sRID1,
                                     const ShortRecordID *sRID2);
```

sRID1	A pointer to the first ShortRecordID structure you want to compare.
sRID2	A pointer to the second ShortRecordID structure you want to compare.

DESCRIPTION

If both ShortRecordID structures are equal, then the OCEEqualShortRecordID function returns true; otherwise, it returns false.

The OCEEqualShortRecordID function compares two ShortRecordID structures for equality in the following manner:

- If both pointers to the ShortRecordID structures are set to nil, then they are equal.
- If one of the pointers to a ShortRecordID structure is set to nil and the other is not, then the OCEEqualShortRecordID function returns false.
- If neither pointer to the ShortRecordID structures is set to nil, then the cid fields of the two ShortRecordID structures are compared for equality by calling the OCEEqualCreationID function (page 2-52). If the OCEEqualCreationID function returns false, then the ShortRecordID structures are not equal.
- If the CreationID fields of the two ShortRecordID structures are equal, then the PackedRLI structures pointed to by the PackedRLIPtr fields of the two ShortRecordID structures are compared for equality by calling the OCEEqualPackedRLI function (page 2-76). If the OCEEqualPackedRLI function returns true, then the two ShortRecordID structures are equal; otherwise, they are not.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0317

SEE ALSO

The `ShortRecordID` structure is defined on page 2-35.

Record Identifier Functions

The functions in this section manipulate record identifier structures. The record identifier is defined by the `RecordID` data structure (page 2-34).

OCEGetIndRecordType

The `OCEGetIndRecordType` function returns a standard record type based on the index value you pass to it.

```
pascal RString *OCEGetIndRecordType
                        (const OCERecordTypeIndex stringIndex);
```

`stringIndex`
One of the index values from the `OCERecordTypeIndex` enumerated list.

DESCRIPTION

To obtain a standard record type, you call the `OCEGetIndRecordType` function and pass it an index value based on the type of record you want. The record type index (page 2-28) is an enumerated list containing all of the standard AOCE record types.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$031B

SEE ALSO

The `recordType` field is part of the `LocalRecordID` structure defined on page 2-27.

For an enumerated list containing all of the standard AOCE record types, see the record type index on page 2-28.

OCENewRecordID

The `OCENewRecordID` function converts data you supply into a `RecordID` structure.

```
pascal void OCENewRecordID(const PackedRLI *theRLI,
                           const LocalRecordID *lRID, RecordID *rid);
```

`theRLI` A pointer to the `PackedRLI` structure you want stored in the `RecordID` structure.

`lRID` A pointer to the `LocalRecordID` structure you want stored in the `RecordID` structure.

`rid` A pointer to the destination `RecordID` structure. You must allocate this structure.

DESCRIPTION

The `OCENewRecordID` function converts a `PackedRLI` structure and `LocalRecordID` structure into a `RecordID` structure.

SPECIAL CONSIDERATIONS

Because the `OCENewRecordID` function does not allocate any memory, the `RecordID` structure it forms uses the same `PackedRLI` structure and the same `LocalRecordID` structure that you supplied as parameters. Therefore, you should not dispose of or reuse the `PackedRLI` and `LocalRecordID` structures until you have finished using the `RecordID` structure as well. Doing so will cause the pointers in the `RecordID` structure to point to incorrect locations in memory and might cause your application to crash.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0320

SEE ALSO

The `RecordID` structure is defined on page 2-34.

OCECopyRecordID

The `OCECopyRecordID` function copies one `RecordID` structure to another.

```
pascal OSErr OCECopyRecordID(const RecordID *rid1,
                             const RecordID *rid2);
```

`rid1`	The source `RecordID` structure.
`rid2`	The destination `RecordID` structure.

DESCRIPTION

Given two `RecordID` structures pointed to by the `rid1` and `rid2` parameters, the `OCECopyRecordID` function copies the contents of the first one into the second. The destination `RecordID` structure must contain pointers to structures large enough to hold copies of the corresponding fields from the source `RecordID` structure; otherwise, a memory-full error is returned. Therefore, when you allocate a new destination `RecordID` structure, you must set the length fields of its `LocalRecordID.recordName`, `LocalRecordId.recordType`, and `LocalRecordID.PackedRLI` fields to their proper values before calling the `OCECopyRecordID` function.

You obtain the correct size for the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields of a `RecordID` structure from the values contained in their `RStringHeader` fields. Once you obtain these values, you can then use them to allocate `recordName` and `recordType` structures of the correct size.

You obtain the correct size for a `LocalRecordId.PackedRLI` structure from the value contained in its `dataLength` field. Once you obtain this value you can then use it to allocate a `PackedRLI` structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0309

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The destination `RecordID` structure is not large enough to hold the contents of the source `RecordID` structure

SEE ALSO

The `RecordID` structure is defined on page 2-34.

OCEEqualRecordID

The `OCEEqualRecordID` function checks the equality of two record identifier structures.

```
pascal Boolean OCEEqualRecordID(const RecordID *rid1,
                                const RecordID *rid2);
```

`rid1`	A pointer to the first `RecordID` you want to compare.
`rid2`	A pointer to the second `RecordID` you want to compare.

DESCRIPTION

The `OCEEqualRecordID` function compares two `RecordID` structures for equality and returns `true` if they are equal, `false` if they are not. This function checks the two `RecordID` structures for equality in the following manner:

- If both pointers to the `RecordID` structures are set to `nil`, then they are equal.
- If one of the pointers to a `RecordID` structure is set to `nil` and the other is not, then the `OCEEqualRecordID` function returns `false`.
- If neither pointer to the `RecordID` structures is set to `nil`, then the `CreationID` structures pointed to by the `LocalRecordID.cid` fields of the two `RecordID` structures are compared for equality by calling the `OCEEqualCreationID` function (page 2-52). If the `OCEEqualCreationID` function returns `false`, then the two `RecordID` structures are not equal.
- If the `CreationID` structures identified by the `LocalRecordID.cid` fields of the two `RecordID` structures are equal, then the `PackedRLI` structures pointed to by the `PackedRLIPtr` fields of the two `RecordID` structures are compared for equality by calling the `OCEEqualPackedRLI` function (page 2-76). If the `OCEEqualPackedRLI` function returns `false`, then the two `RecordID` structures are not equal.
- If the `PackedRLI` structures pointed to by the `PackedRLIPtr` fields of the two `RecordID` structures are equal, then the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields of the two `RecordID` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) function and passing it the proper `RStringKind` value for each field. If the `OCEEqualRString` function returns `false`, the two `RecordID` structures are not equal.

If the conditions for equality listed above are satisfied, then the two `RecordID` structures are equal; otherwise, they are not equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0314

SEE ALSO

The `RecordID` structure is defined on page 2-34.

The `RStringKind` structure is defined on page 2-24.

Packed Record Identifier Functions

The functions described in this section manipulate packed record identifiers. Packed record identifiers are defined by the `PackedRecordID` structure (page 2-35).

OCECopyPackedRecordID

The `OCECopyPackedRecordID` function copies one `PackedRecordID` structure to another.

```
pascal OSErr OCECopyPackedRecordID(const PackedRecordID *pRID1,
                                   const PackedRecordID *pRID2,
                                   unsigned short pRID2Length);
```

`pRID1` A pointer to the source `PackedRecordID` structure.

`pRID2` A pointer to the destination `PackedRecordID` structure.

`pRID2Length`
The length, in bytes, of the destination `PackedRecordID` structure, not including the bytes in the `dataLength` field.

DESCRIPTION

Given two `PackedRecordID` structures pointed to by the `pRID1` and `pRID2` parameters, the `OCECopyPackedRecordID` function copies the contents of the first into the second. The `pRID2Length` parameter is the size of the destination `PackedRecordID` structure, excluding its `dataLength` field. The destination `PackedRecordID` structure must be large enough to hold the entire contents of the source `PackedRecordID` structure; otherwise, a memory-full error is returned.

You obtain the proper size for a `PackedRecordID` structure from the value contained in its `dataLength` field. Once you obtain this value, you can then use it to allocate a destination `PackedRecordID` structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0306

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The `pRID2` parameter is not large enough to hold the entire contents of `pRID1`

SEE ALSO

The `PackedRecordID` structure is defined on page 2-35.

OCEPackedRecordIDSize

The `OCEPackedRecordIDSize` function computes the number of bytes of memory needed to hold a `PackedRecordID` structure.

```
pascal unsigned short OCEPackedRecordIDSize(const RecordID *rid);
```

`rid` A pointer to a `RecordID` structure.

DESCRIPTION

The `OCEPackedRecordIDSize` function returns the number of bytes that a `PackedRecordID` needs to hold the packed data from a specified `RecordID` structure. The number of bytes returned by the `OCEPackedRecordIDSize` function includes the size of the `datalength` field of the `PackedRecordID` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032B

SEE ALSO

The `RecordID` structure is defined on page 2-34.

The `PackedRecordID` structure is defined on page 2-35.

To unpack a `PackedRecordID` structure into a `RecordID` structure, use the `OCEUnpackRecordID` function (page 2-91).

OCEPackRecordID

The `OCEPackRecordID` function packs a `RecordID` structure into a `PackedRecordID` structure.

```
pascal OSErr OCEPackRecordID(const RecordID *rid,
                             PackedRecordID *pRID,
                             unsigned short packedRecordIDLength);
```

`rid` A pointer to the `RecordID` structure you want packed.

`pRID` A pointer to a `PackedRecordID` structure. You must allocate this structure.

`packedRecordIDLength`
The maximum length, in bytes, of the `PackedRecordID` structure, excluding the bytes in the `dataLength` field.

DESCRIPTION

The `OCEPackRecordID` function packs a `RecordID` structure into a `PackedRecordID` structure. The `PackedRecordID` structure must be large enough to contain the entire contents of the `RecordID` in packed format; otherwise, a memory-full error is returned. You obtain the size of a `PackedRecordID` structure large enough to hold the data in a `RecordID` structure by calling the `OCEPackedRecordIDSize` function described on page 2-90.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0326

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The `PackedRecordID` structure is not large enough to hold the packed data from the `RecordID` structure

SEE ALSO

The `RecordID` structure is defined on page 2-34.

The `PackedRecordID` structure is defined on page 2-35.

To unpack a `PackedRecordID` structure into a `RecordID` structure, see the `OCEUnpackRecordID` function, described next.

OCEUnpackRecordID

The `OCEUnpackRecordID` function unpacks a `PackedRecordID` structure into a `RecordID` structure.

```
pascal void OCEUnpackRecordID(const PackedRecordID *pRID,
                              RecordID *rid);
```

`pRID`	A pointer to the `PackedRecordID` structure you want to unpack.
`rid`	A pointer to a `RecordID` structure. You must allocate this structure.

DESCRIPTION

Given a `PackedRecordID` structure pointed to by the `pRID` parameter and a `RecordID` structure pointed to by the `rid` parameter, the `OCEUnpackRecordID` function unpacks the `PackedRecordID` structure into the `RecordID` structure.

SPECIAL CONSIDERATIONS

Because the `OCEUnpackRecordID` function does not allocate any memory, the unpacked `RecordID` structure contains pointers into the `PackedRecordID` structure. Therefore, do not delete or reuse the `PackedRecordID` structure until you have finished using the unpacked `RecordID` structure as well. Doing so will cause the pointers in the `RecordID` structure to point to incorrect locations in memory, and your application may crash when you try to access the `RecordID` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0332

SEE ALSO

The `RecordID` structure is defined on page 2-34.

The `PackedRecordID` structure is defined on page 2-35.

To pack a `RecordID` structure into a `PackedRecordID` structure, see the `OCEPackRecordID` function described on page 2-90.

OCEEqualPackedRecordID

The `OCEEqualPackedRecordID` function checks the equality of two `PackedRecordID` structures.

```
pascal Boolean OCEEqualPackedRecordID
                              (const PackedRecordID *pRID1,
                               const PackedRecordID *pRID2);
```

`pRID1` A pointer to the first `PackedRecordID` structure you want to compare.

`pRID2` A pointer to the second `PackedRecordID` structure you want to compare.

DESCRIPTION

The `OCEEqualPackedRecordID` function compares two `PackedRecordID` structures for equality and returns `true` if they are equal and `false` if they are not.

This function checks the two `PackedRecordID` structures for equality in the following manner:

- If both pointers to the `PackedRecordID` structures are `nil`, then they are equal.
- If one of the pointers to a `PackedRecordID` structure is `nil` and the other is not, then the `PackedRecordID` structures are not equal.

- If neither pointer to the `PackedRecordID` structures is `nil`, then they are unpacked and the `CreationID` structures identified by the `LocalRecordId.cid` fields of the two unpacked `PackedRecordID` structures are compared for equality by calling the `OCEEqualCreationID` function (page 2-52). If the `OCEEqualCreationID` function returns `false`, then the two `PackedRecordID` structures are not equal.
- If the `CreationID` structures identified by the `LocalRecordId.cid` fields of the two unpacked `PackedRecordID` structures are equal, then the `PackedRLI` structures pointed to by the `PackedRLIPtr` fields of the two `PackedRecordID` structures are compared for equality by calling the `OCEEqualPackedRLI` function (page 2-76). If the `OCEEqualPackedRLI` function returns `false`, then the two `PackedRecordID` structures are not equal.
- If the `PackedRLI` structures pointed to by the `PackedRLIPtr` fields of the two (unpacked) `PackedRecordID` structures are equal, then the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields of the two (unpacked) `PackedRecordID` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) function and passing it the proper `RStringKind` value for each field. If the `OCEEqualRString` function returns `false`, the two `PackedRecordID` structures are not equal.

If the conditions for equality listed above are satisfied, then the two `PackedRecordID` structures are equal; otherwise, they are not equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0312

SEE ALSO

The `PackedRecordID` structure is defined on page 2-35.

The `RStringKind` structure is defined on page 2-24.

OCEValidPackedRecordID

The `OCEValidPackedRecordID` function checks the validity of a packed record identifier.

```
pascal Boolean OCEValidPackedRecordID(const PackedRecordID *pRID);
```

`pRID` A pointer to the `PackedRecordID` you want to validate.

DESCRIPTION

Given a pointer to a `PackedRecordID` structure, the `OCEValidPackedRecordID` function checks it for validity based on its internal structure and returns `true` if it is valid and `false` if it is not. The `OCEValidPackedRecordID` function checks a `PackedRecordID` structure for validity in the following manner:

- If the pointer to the `PackedRecordID` structure is set to `nil`, or the length of the `PackedRecordID` structure is 0, then the `PackedRecordID` structure is invalid.
- If the pointer to the `PackedRecordID` structure is not `nil` and the length of the structure is greater than 0, then it is unpacked and the `RLI` component of the `PackedRecordID` structure is validated by calling the `OCEValidRLI` function (page 2-69). If the `OCEValidRLI` function returns `false`, then the `PackedRecordID` structure is not valid.
- If the `RLI` component of the `PackedRecordID` structure is valid, then the `recordName` and `recordType` fields of the `PackedRecordID` structure are validated by calling the `OCEValidRString` function. If the `OCEValidRString` function returns `false`, then the `PackedRecordID` structure is not valid.
- If all of the conditions tested for validity are `true`, then the entire `PackedRecordID` structure is valid.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0335

SEE ALSO

The `PackedRecordID` structure is defined on page 2-35.

Attribute Type Functions

The function described in this section returns a standard attribute type. The attribute type is defined by the `AttributeType` data structure and is described on page 2-39.

OCEGetIndAttributeType

The `OCEGetIndAttributeType` function returns an attribute type based on the index value you pass to it.

```
pascal AttributeType *OCEGetIndAttributeType(const
                                OCEAttributeTypeIndex stringIndex);
```

`stringIndex`
One of the index values from the `OCEAttributeTypeIndex` enumerated list.

DESCRIPTION

To obtain a standard attribute type, you call the `OCEGetIndAttributeType` function and pass it an index value based on the kind of attribute type you want. The attribute type index (page 2-40) is an enumerated list containing all of the standard AOCE attribute types.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$031A

SEE ALSO

The `AttributeType` structure is defined on page 2-39.

For an enumerated list of all the standard AOCE attribute types, see the attribute type index on page 2-40.

Catalog Services Specification Functions

The functions described in this section manipulate the various catalog services specification data structures. The catalog services specification is defined by the `DSSpec` data structure (page 2-36) and its packed form by the `PackedDSspec` structure (page 2-37). These functions perform such tasks as copying, comparing, unpacking, and retrieving information from `DSSpec` structures.

Other forms of the `DSSpec` structure include the `OCERecipient` and the packed form, `OCEPackedRecipient`, which are defined in the chapter "Interprogram Messaging Manager" in this book. The functions, such as `OCEPackRecipient`, that manipulate these data structures are also described in the chapter "Interprogram Messaging Manager."

OCECopyPackedDSSpec

The `OCECopyPackedDSSpec` function copies data from one `PackedDSSpec` into another.

```
pascal OSErr OCECopyPackedDSSpec(const PackedDSSpec *pdss1,
         const PackedDSSpec *pdss2, unsigned short pdss2Length);
```

`pdss1` A pointer to the source `PackedDSSpec` structure.

`pdss2` A pointer to the destination `PackedDSSpec` structure.

`pdss2Length`
The length, in bytes, of the destination `PackedDSSpec` structure, not including the header information.

DESCRIPTION

Given two `PackedDSSpec` structures pointed to by the `pdss1` and `pdss2` parameters, the `OCECopyPackedDSSpec` function copies the first into the second. The `pdss2Length` parameter is the size, in bytes, of the destination `PackedDSSpec` structure, excluding its header. The destination `PackedDSSpec` structure must be large enough to hold the entire contents of the source `PackedDSSpec` structure; otherwise, a memory-full error is returned.

You obtain the proper size for a `PackedDSSpec` structure from the value contained in its `dataLength` field. Once you obtain this value, you can then use it to allocate a destination `PackedDSSpec` structure of the correct size.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0303

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	The destination `PackedDSSpec` structure is not large enough to hold the contents of the source `PackedDSSpec` structure

SEE ALSO

The `PackedDSSpec` data structure is defined on page 2-37.

OCEPackedDSSpecSize

The `OCEPackedDSSpecSize` function computes the number of bytes of memory needed to hold a packed `DSSpec` structure.

```
pascal unsigned short OCEPackedDSSpecSize(const DSSpec *dss);
```

`dss` A pointer to the `DSSpec` structure whose size, when packed, you want to determine.

DESCRIPTION

The `OCEPackedDSSpecSize` function computes the number of bytes required to hold the information contained in a `DSSpec` structure when it is packed. The number of bytes returned by the `OCEPackedDSSpecSize` function includes the `dataLength` field of the `PackedDSSpec` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0327

SEE ALSO

The `DSSpec` structure is defined on page 2-36.

The `PackedDSSpec` structure is defined on page 2-37.

To pack a `DSSpec` structure, use the `OCEPackDSSpec` function, described next.

OCEPackDSSpec

The `OCEPackDSSpec` function forms a `PackedDSSpec` structure from a `DSSpec` structure.

```
pascal OSErr OCEPackDSSpec(const DSSpec *dss, PackedDSSpec *pdss,
                           unsigned short pdssLength);
```

`dss` A pointer to the `DSSpec` structure that you want to pack.

`pdss` A pointer to a `PackedDSSpec` structure. You must allocate this structure.

`pdssLength` The maximum number of bytes that can be stored in the `PackedDSSpec` structure, not including the header information.

DESCRIPTION

The `OCEPackDSSpec` function packs the contents of a `DSSpec` structure into a `PackedDSSpec` structure. The `PackedDSSpec` structure must be large enough to contain the packed `RecordID` information and any extension value as well; otherwise, a memory-full error is returned. Use the `OCEPackDSSpecSize` function to obtain the size of a `PackedDSSpec` structure.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0322

RESULT CODES

noErr	0	No error
memFullErr	–108	The PackedDSSpec structure is not large enough to hold all of the packed information

SEE ALSO

The DSSpec structure is defined on page 2-36.

The PackedDSSpec structure is defined on page 2-37.

To obtain the size of a PackedDSSpec structure, use the OCEPackDSSpecSize function on page 2-96.

For information on unpacking a PackedDSSpec structure, see the OCEUnpackDSSpec function, described next.

OCEUnpackDSSpec

The OCEUnpackDSSpec function unpacks a PackedDSSpec structure.

```
pascal void OCEUnpackDSSpec(const PackedDSSpec *pdss, DSSpec *dss,
RecordID *rid);
```

pdss	A pointer to the PackedDSSpec structure you want to unpack.
dss	A pointer to a DSSpec structure. You must allocate this structure.
rid	A pointer to a RecordID structure. The OCEUnpackDSSpec function extracts the RecordID information from the PackedDSSpec structure and places it in this RecordID structure. You must allocate this structure.

DESCRIPTION

The OCEUnpackDSSpec function extracts the information from a PackedDSSpec structure and places it in a DSSpec structure and a RecordID structure. The OCEUnpackDSSpec function extracts the record identifier (if any) into the RecordID structure, places the rest of the information into the DSSpec structure, and then sets the entitySpecifier field of the DSSpec structure to point to the RecordID structure. The OCEUnpackDSSpec function returns a pointer to the extension (if any) in the extensionValue field of the DSSpec structure, and returns the length of that extension in the extensionSize field of the DSSpec structure. If there is no extension, the OCEUnpackDSSpec function sets the extensionValue field of the DSSpec structure to nil.

SPECIAL CONSIDERATIONS

The unpacked `DSSpec` and `RecordID` structures contain pointers into the `PackedDSSpec` structure. You should not delete or reuse the `PackedDSSpec` structure until you have finished using the `DSSpec` and `RecordID` structures as well.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$032F

SEE ALSO

The `DSSpec` structure is defined on page 2-36.

The `PackedDSSpec` structure is defined on page 2-37.

The `RecordID` structure is defined on page 2-34.

To pack a `DSSpec` structure, use the `OCEPackDSSpec` function (page 2-97).

OCEEqualDSSpec

The `OCEEqualDSSpec` function checks the equality of two `DSSpec` structures.

```
pascal Boolean OCEEqualDSSpec(const DSSpec *pdss1,
                              const DSSpec *pdss2);
```

`pdss1` A pointer to the first `DSSpec` structure you want to compare.

`pdss2` A pointer to the second `DSSpec` structure you want to compare.

DESCRIPTION

Given two `DSSpec` structures pointed to by the `pdss1` and `pdss2` parameters, the `OCEEqualDSSpec` function compares them for equality and returns `true` if they are equal and `false` if they are not.

This function checks the two `DSSpec` structures for equality in the following manner:

- If both pointers to the `DSSpec` structures are `nil`, then they are equal.
- If one of the pointers to a `DSSpec` structure is `nil` and the other is not, then the two `DSSpec` structures are not equal.
- If neither pointer to the `DSSpec` structures is `nil`, then the `CreationID` structures, identified by the `RecordID->LocalRecordID.cid` fields of the two `DSSpec` structures, are compared for equality by calling the `OCEEqualCreationID` function (page 2-52). If the `OCEEqualCreationID` function returns `false`, then the two `DSSpec` structures are not equal.

- If the `CreationID` structures are equal, then the `PackedRLI` structures pointed to by the `RecordID->PackedRLIPtr` fields of the two `DSSpec` structures are compared for equality by calling the `OCEEqualPackedRLI` function (page 2-76). If the `OCEEqualPackedRLI` function returns `false`, then the two `DSSpec` structures are not equal.
- If the `PackedRLI` structures are equal, then the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields belonging to the `RecordID` structure of the two `DSSpec` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) function and passing it the proper `RStringKind` value for each field. If the `OCEEqualRString` function returns `false`, the two `DSSpec` structures are not equal.
- If the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields are equal then the values of the `extensionType` fields of the `DSSpec` structures are examined. If they are not identical then the `DSSpec` structures are not equal.
- If the `extensionType` fields of the two `DSSpec` structures are identical, then the `extensionSize` fields of the `DSSpec` structures are compared. If they are not identical, then the two `DSSpec` structures are not equal.
- If the `extensionSize` fields of the two `DSSpec` structures are identical, then the `extensionValue` fields of the `DSSpec` structures are compared. They are compared byte by byte for equality, and if they are not identical then the two `DSSpec` structures are not equal.

If the conditions for equality listed above are satisfied, then the two `DSSpec` structures are equal; otherwise, they are not equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$030E

SEE ALSO

The `DSSpec` data structure is defined on page 2-36.

To compare two `PackedDSSpec` structures for equality use the `OCEEqualPackedDSSpec` function, described next.

OCEEqualPackedDSSpec

The `OCEEqualPackedDSSpec` function checks the equality of two `PackedDSSpec` structures.

```
pascal Boolean OCEEqualPackedDSSpec(const PackedDSSpec *pdss1,
                                    const PackedDSSpec *pdss2);
```

`pdss1`	A pointer to the first `PackedDSSpec` structure you want to compare.
`pdss2`	A pointer to the second `PackedDSSpec` structure you want to compare.

DESCRIPTION

Given two `PackedDSSpec` structures pointed to by the `pdss1` and `pdss2` parameters, the `OCEEqualPackedDSSpec` function compares them for equality and returns `true` if they are equal, and `false` if they are not.

This function checks the two `PackedDSSpec` structures for equality in the following manner:

- If both pointers to the `PackedDSSpec` structures are `nil`, then they are equal.
- If one of the pointers to a `PackedDSSpec` structure to `nil` and the other is not, then the two `PackedDSSpec` structures are not equal.
- If neither pointer to the `PackedDSSpec` structures is `nil`, then the two structures are unpacked and the `CreationID` structures, identified by the `RecordID->LocalRecordID.cid` fields of the two `DSSpec` structures, are compared for equality by calling the `OCEEqualCreationID` function (page 2-52). If the `OCEEqualCreationID` function returns `false`, then the two `PackedDSSpec` structures are not equal.
- If the `CreationID` structures are equal, then the `PackedRLI` structures pointed to by the `RecordID->PackedRLIPtr` fields of the two unpacked `PackedDSSpec` structures are compared for equality by calling the `OCEEqualPackedRLI` function (page 2-76). If the `OCEEqualPackedRLI` function returns `false`, then the two `PackedDSSpec` structures are not equal.
- If the `PackedRLI` structures are equal, then the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields belonging to the `RecordID` structure of the two unpacked `PackedDSSpec` structures are compared for equality by calling the `OCEEqualRString` (page 2-50) function and passing it the proper `RStringKind` value for each field. If the `OCEEqualRString` function returns `false`, the two `PackedDSSpec` structures are not equal.
- If the `LocalRecordID.recordName` and `LocalRecordID.recordType` fields belonging to the `RecordID` structure of the two unpacked `PackedDSSpec` structures are equal then the values of the `extensionType` fields of the `PackedDSSpec` structures are examined. If they are not identical then the `PackedDSSpec` structures are not equal.
- If the `extensionType` fields of the two unpacked `PackedDSSpec` structures are identical, then the `extensionSize` fields of the unpacked `PackedDSSpec` structures are compared. If they are not identical, then the two `PackedDSSpec` structures are not equal.
- If the `extensionSize` fields of the two unpacked `PackedDSSpec` structures are identical, then the `extensionValue` fields of the unpacked `PackedDSSpec` structures are compared. They are compared byte by byte for equality, and if they are not identical then the two `PackedDSSpec` structures are not equal.

If the conditions for equality listed above are satisfied, then the two `PackedDSSpec` structures are equal; otherwise, they are not equal.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0310

SEE ALSO

The `PackedDSSpec` data structure is defined on page 2-37.

The `RStringKind` data structure is defined on page 2-24.

To compare two `DSSpec` structures for equality use the `OCEEqualDSSpec` function, described on page 2-99.

OCEValidPackedDSSpec

The `OCEValidPackedDSSpec` function checks the validity of a `PackedDSSpec` structure.

```
pascal Boolean OCEValidPackedDSSpec(const PackedDSSpec *pdss);
```

`pdss` A pointer to the `PackedDSSpec` that you want to verify.

DESCRIPTION

The `OCEValidPackedDSSpec` function examines a `PackedDSSpec` structure to ensure validity for its particular type and returns `true` if it is valid, `false` if it is not.

The `OCEValidPackedDSSpec` function determines validity for a `PackedDSSpec` structure in the following manner:

- If the pointer to the `PackedDSSpec` structure is `nil`, then the `PackedDSSpec` structure is invalid.
- If the length of the `PackedDSSpec` structure is 0, then the `PackedDSSpec` structure is valid.
- If the pointer to the `PackedDSSpec` structure is not `nil`, and the length of the `PackedDSSpec` structure is greater than 0, then the `PackedDSSpec` structure is unpacked and its `extensionType` field is examined for validity. If the `extensionType` field of the `PackedDSSpec` has a value of `'entn'`, then the `OCEValidPackedDSSpec` function checks to make sure that the `PackedDSSpec` structure contains a valid `entitySpecifier` field by calling the `OCEValidPackedRecordID` function (page 2-93). If the `OCEValidPackedRecordID` function returns `false`, the `PackedDSSpec` structure is not valid.
- If the `extensionType` field does not have a value of '`entn`' and it is not `nil`, then the `RecordID` field of the `PackedDSSpec` is examined to see if it contains an `RLI` component. If it does, then the `RLI` structure is checked for validity by calling the `OCEValidRLI` function. If the `OCEValidRLI` function returns `false`, then the

PackedDSSpec structure is invalid. The CreationID structure, identified by the RecordID->LocalRecordID.cid field of the unpacked PackedDSSpec structure, is not tested for validity.

- If the RLI component of the PackedDSSpec is valid, then the LocalRecordID.recordName and LocalRecordID.recordType fields of the RecordID component of the PackedDSSpec structure are examined for validity by calling the OCEValidRString function (page 2-51). If the OCEValidRString function returns false, then the PackedDSSpec is invalid.

If all of the conditions for validity are satisfied, then the PackedDSSpec structure is valid; otherwise, it is not.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
__OCEUtils	$0333

SEE ALSO

The PackedDSSpec data structure is defined on page 2-37.

OCEGetDSSpecInfo

The OCEGetDSSpecInfo function returns information about a DSSpec structure.

```
pascal OSType OCEGetDSSpecInfo(const DSSpec *spec);
```

spec A pointer to the DSSpec structure you want to get information about.

DESCRIPTION

The OCEGetDSSpecInfo function returns certain information about the specific DSSpec structure you pass to it. If the DSSpec structure does not have an entity specifier, it is invalid, that is, it returns kOCEInvalidDSSpec. If it does have an entity specifier, then it must have an extension type value of 'entn'; otherwise, it is invalid.

If the DSSpec structure has no extension, the OCEGetDSSpecInfo function determines whether it represents the root of all catalogs, a single catalog, a catalog node, or a record. If it has no extension and is not any of these types, it is considered invalid. If the DSSpec structure does have an extension, this function simply returns the extension type. The OCEGetDSSpecInfo function only performs the rudimentary checks just described. It does not do a complete check of the DSSpec structure for validity. Call the OCEValidPackedDSSpec function (page 2-102) to check a PackedDSSpec structure for validity.

The values that are returned by the `OCEGetDSSpecInfo` function are described in this enumerated list:

```
enum /* OCEGetDSSpecInfo types */
{
   kOCEInvalidDSSpec= '0x3F3F3F3FL',/* could not be determined */
   kOCEDirsRootDSSpec= 'root',      /* root of all catalogs
                                       ("Catalog" icon) */
   kOCEDirectoryDSSpec= 'dire',     /* catalog */
   kOCEDNodeDSSpec= 'dnod',         /* dNode */
   kOCERecordDSSpec= 'reco',        /* record */
   kOCEentnDSSpec= 'entn',          /* extensionType is 'entn' */
   kOCENOTentnDSSpec= 'not '        /* extensionType is
                                          not 'entn' */
};
```

Field descriptions

`kOCEInvalidDSSpec`
The type does not conform to any known type.

`kOCEDirsRootDSSpec`
The `DSSpec` structure represents the root of all catalogs.

`kOCEDirectoryDSSpec`
The `DSSpec` structure represents a catalog.

`kOCEDNodeDSSpec`
The `DSSpec` structure represents a catalog node.

`kOCERecordDSSpec`
The `DSSpec` structure represents a record.

`kOCEentnDSSpec`
The extension type of the `DSSpec` structure is `'entn'`.

`kOCENOTentnDSSpec`
The `entitySpecifier` field of the `DSSpec` structure is not `nil` and the extension type of the `DSSpec` structure is not `'entn'`.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$0319

SEE ALSO

The `DSSpec` data structure is defined on page 2-36.

To obtain the extension type of a `DSSpec` structure, use the `OCEGetExtensionType` function, described next.

OCEGetExtensionType

The `OCEGetExtensionType` function returns the extension type embedded in a `PackedDSSpec` structure.

```
pascal OSType OCEGetExtensionType(const PackedDSSpec *pdss);
```

`pdss` A pointer to a `PackedDSSpec` structure from which you want to retrieve the extension type.

DESCRIPTION

Given a pointer to a `PackedDSSpec` structure, the `OCEGetExtensionType` function extracts the extension type of the `PackedDSSpec` structure and returns it to you.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$031C

SEE ALSO

The `DSSpec` data structure is defined on page 2-36.

To obtain information about a `DSSpec` structure, see the `OCEGetDSSpecInfo` function on page 2-103.

OCEStreamPackedDSSpec

The `OCEStreamPackedDSSpec` function takes a `DSSpec` structure and converts it from a pointer-based structure into a stream of bytes.

```
pascal OSErr OCEStreamPackedDSSpec(const DSSpec *dss,
                                   MyDSSpecStreamer stream,
                                   long userData,
                                   unsigned long *actualCount);
```

`dss` A pointer to the `DSSpec` structure you want to process.

`stream` A pointer to a function that you supply.

`userData` Data supplied by you that is passed on to your `stream` function. The `userData` parameter can contain anything your particular stream method needs.

`actualCount`
A pointer to the total number of bytes (streamed out) by the `OCEStreamPackedDSSpec` function.

DESCRIPTION

The `OCEStreamPackedDSSpec` function converts a `DSSpec` structure into a stream of bytes by calling the `stream` function that you provide. You can use this function whenever you want to write the contents of a `DSSpec` structure as a series of bytes to a file, into a buffer in memory, or any other place.

The `stream` function that you provide contains the specific code that writes out the data. The `OCEStreamPackedDSSpec` function calls your `stream` function repeatedly and passes your function the current portion of the data that needs to be streamed, the length of this data, an `eof` flag that is set by the `OCEStreamPackedDSSpec` function if this is the last of the data to be streamed, and a parameter containing any application-specific data that you define. For example, if you were writing a `stream` function that wrote out a `DSSpec` structure to a file on a hard disk, you might want to store a pointer in the `userData` parameter to a block of data that contains such information as the filename and size of the file.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`__OCEUtils`	$033D

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `DSSpec` data structure is defined on page 2-36.

The callback function `MyDSSpecStreamer` is described next.

Application-Defined Functions

This section describes a callback function that you supply to `OCEStreamPackedDSSpec`. See the section "Application-Defined Functions" in the chapter "Catalog Manager" in this book for more information on how AOCE callback functions operate.

MyDSSpecStreamer

The `MyDSSpecStreamer` function provides a method for processing data from the `OCEStreamPackedDSSpec` function.

```
typedef pascal OSErr (*MyDSSpecStreamer)(void *buffer,
                     unsigned long count, Boolean eof ,
                     long userData);
```

`buffer`	A pointer to the data that your streamer method processes. This is supplied by the `OCEStreamPackedDSSpec` function each time it calls your `MyDSSpecStreamer` function.
`count`	The length, in bytes, of the current data in the buffer.
`eof`	A flag that is set by the `OCEStreamPackedDSSpec` function the last time that it calls your `MyDSSpecStreamer` function. This flag informs you that when the `OCEStreamPackedDSSpec` function finishes processing the data currently in the buffer, it will have completed processing the `DSSpec` structure.
`userData`	The data that you supply in the `userData` parameter to the `OCEStreamPackedDSSpec` function. This is passed directly to your `MyDSSpecStreamer` function.

DESCRIPTION

The `MyDSSpecStreamer` function is called by the `OCEStreamPackedDSSpec` function (page 2-105) to process the data from a `DSSpec` structure in discreet segments. You write this routine to process the data in the way that you want. The `OCEStreamPackedDSSpec` function calls your `MyDSSpecStreamer` function various times and passes your function progress information as well as the current portion of the `DSSpec` to process. Any errors returned by this function are passed on to the `OCEStreamPackedDSSpec` function.

SEE ALSO

The `DSSpec` data structure is defined on page 2-36.

The `OCEStreamPackedDSSpec` function is defined on page 2-105.

Summary of the AOCE Utilities

C Summary

Constants and Data Types

```
/* OCE String Constants */
#define RStringHeader \
   CharacterSet charSet;\
   unsigned short dataLength;

enum {
   kRString32Size          = 32,       /* max size of RString32 */
   kRString64Size          = 64,       /* max size of RString64 */
   kNetworkSpecMaxBytes    = 32,       /* max size of NetworkSpec */
   kPathNameMaxBytes       = 1024,     /* max size of PackedPathName */
   kDirectoryNameMaxBytes  = 32,       /* max size of DirectoryName */
   kAttributeTypeMaxBytes  = 32,       /* max size of AttributeType */
   kAttrValueMaxBytes      = 65536,    /* max size of any attribute value */
   kRStringMaxBytes        = 256,      /* max size of recordName or
                                          recordType */
   kRStringMaxChars        = 128       /* max # of chars in recordName
                                          RString, or recordType */
};

#define kMinPackedRStringLength (sizeof (ProtoRString))

/* RStringKind Values */
enum {
   kOCEDirName             = 0,        /* RString is a Catalog Name */
   kOCERecordOrDNodeName   = 1,        /* RString is a recordName or
                                          catalog node name */
   kOCERecordType          = 2,        /* RString is a recordType */
   kOCENetworkSpec         = 3,        /* RString is a NetworkSpec */
   kOCEAttrType            = 4,        /* RString is an AttributeType */
   kOCEGenericSensitive    = 5,        /* RString is a case and diacritical
                                          mark sensitive generic string */
```

```
   kOCEGenericInsensitive  = 6         /* RString is a case and diacritical
                                          mark insensitive generic string */
};

/* OCEDirectoryKind Values */
enum {
   kDirAllKinds            = 0,        /* All catalog types */
   kDirADAPKind            = 'adap',   /* an PowerShare catalog */
   kDirPersonalDirectoryKind
                           = 'pdir',   /* a personal catalog */
   kDirDSAMKind            = 'dsam'    /* catalog service access module */
}

/* Catalog Node Constants */
enum {
   kNULLDNodeNumber        = 0,        /* none specified */
   kRootDNodeNumber        = 2         /* the root of the tree */
};

/* Values returned by OCEGetDSSpecInfo() */
enum {
   kOCEInvalidDSSpec       = 0x3F3F3F3FL,/* '????' could not be
                                            determined */
   kOCEDirsRootDSSpec      = 'root',   /* root of all catalogs
                                          ("Catalog" icon) */
   kOCEDirectoryDSSpec     = 'dire',   /* catalog */
   kOCEDNodeDSSpec         = 'dnod',   /* Dnode */
   kOCERecordDSSpec        = 'reco',   /* record */
   kOCEentnDSSpec          = 'entn',   /* extensionType is 'entn' */
   kOCENOTentnDSSpec       = 'not '    /* extensionType is not 'entn' */
};

/* AttributeTag values */
enum {
   typeRString             = 'rstr',   /* attribute value is an RString */
   typePackedDSSpec        = 'dspc',   /* attribute value is a DSSpec */
   typeBinary              = 'bnry'    /* attribute value is a sequence
                                          of bytes */
};
```

```
/* Cluster info */
enum {
   kcanContainRecordsBit,              /* a cluster */
   kForeignNodeBit                     /* a foreign catalog */
};

/* DirNodeKind */
enum {
   kcanContainRecords= 1L<<kcanContainRecordsBit,
   kForeignNode= 1L<<kForeignNodeBit
};

/* RLI Constants */

#define kMinPackedRLISize (sizeof (ProtoPackedRLI) + \
               sizeof (DirDiscriminator) + sizeof (DNodeNum) +\
               kMinPackedRStringLength + sizeof (ProtoPackedPathName))

#define kRLIMaxBytes (sizeof (RString) + sizeof (DirDiscriminator) + \
                        sizeof (DNodeNum) + kPathNameMaxBytes)

#define PackedRLIHeader unsigned short dataLength /* number of bytes
                                                     in data field */

/* RecordID Constants */

#define kPackedRecordIDMaxBytes (kPathNameMaxBytes + sizeof (DNodeNum) + \
                        sizeof (DirDiscriminator) + sizeof (CreationID) + \
                        (3 * sizeof (RString)))

#define PackedRecordIDHeader unsigned short dataLength /* length of data field
                                          in the PackedRecordID structure */

/* DSSpec Constants */

#define kPackedDSSpecMaxBytes(sizeof (PackedRecordID) + sizeof (OSType) + \
         sizeof (unsigned short))

#define PackedDSSpecHeader unsigned short dataLength;

/* Indices for the standard definitions for standard record types */

#define  kUserRecTypeNum               1     /* "User" */
#define  kGroupRecTypeNum              2     /* "Group" */
#define  kMnMRecTypeNum                3     /* "AppleMail™ M&M" */
#define  kMnMForwarderRecTypeNum       4     /* "AppleMail™ Fwdr" */
```

```
#define  kNetworkSpecRecTypeNum        5     /* "NetworkSpec" */
#define  kADAPServerRecTypeNum         6     /* "PowerShare Server" */
#define  kADAPDNodeRecTypeNum          7     /* "PowerShare DNode" */
#define  kADAPDNodeRepRecTypeNum       8     /* "PowerShare DNode Rep" */
#define  kServerSetupRecTypeNum        9     /* "Server Setup" */
#define  kDirectoryRecTypeNum          10    /* "Catalog" */
#define  kDNodeRecTypeNum              11    /* "DNode" */
#define  kSetupRecTypeNum              12    /* "Setup" */
#define  kMSAMRecTypeNum               13    /* "MSAM" */
#define  kDSAMRecTypeNum               14    /* "CSAM" */
#define  kAttributeValueRecTypeNum     15    /* "Attribute Value" */
#define  kBusinessCardRecTypeNum       16    /* "Business Card" */
#define  kMailServiceRecTypeNum        17    /* "Mail Service" */
#define  kCombinedRecTypeNum           18    /* "Combined" */
#define  kOtherServiceRecTypeNum       19    /* "Other Service" */
#define  kAFPServiceRecTypeNum         20    /* "Other Service afps" */

#define  kFirstOCERecTypeNum kUserRecTypeNum       /* first standard OCE
                                                      record type */

#define  kLastOCERecTypeNum kAFPServiceRecTypeNum/* last standard OCE
                                                      record type */

#define  kNumOCERecTypes      (kLastOCERecTypeNum - kFirstOCERecTypeNum + 1)

/* Indices for the standard definitions for standard attribute types
(OCEAttributeTypeIndex): */

#define  kMemberAttrTypeNum            1001  /* "Member" */
#define  kAdminsAttrTypeNum            1002  /* "Administrators" */
#define  kMailSlotsAttrTypeNum         1003  /* "mailslots" */
#define  kPrefMailAttrTypeNum          1004  /* "pref mailslot" */
#define  kAddressAttrTypeNum           1005  /* "Address" */
#define  kPictureAttrTypeNum           1006  /* "Picture" */
#define  kAuthKeyAttrTypeNum           1007  /* "auth key" */
#define  kTelephoneAttrTypeNum         1008  /* "Telephone" */
#define  kNBPNameAttrTypeNum           1009  /* "NBP Name" */
#define  kQMappingAttrTypeNum          1010  /* "ForwarderQMap" */
#define  kDialupSlotAttrTypeNum        1011  /* "DialupSlotInfo" */
#define  kHomeNetAttrTypeNum           1012  /* "Home Internet" */
#define  kCoResAttrTypeNum             1013  /* "Co-resident M&M" */
#define  kFwdrLocalAttrTypeNum         1014  /* "FwdrLocalRecord" */
#define  kConnectAttrTypeNum           1015  /* "Connected To" */
#define  kForeignAttrTypeNum           1016  /* "Foreign RLIs" */
```

```
#define  kOwnersAttrTypeNum            1017  /* "Owners" */
#define  kReadListAttrTypeNum          1018  /* "ReadList" */
#define  kWriteListAttrTypeNum         1019  /* "WriteList" */
#define  kDescriptorAttrTypeNum        1020  /* "Descriptor" */
#define  kCertificateAttrTypeNum       1021  /* "Certificate" */
#define  kMsgQsAttrTypeNum             1022  /* "MessageQs" */
#define  kPrefMsgQAttrTypeNum          1023  /* "PrefMessageQ" */
#define  kMasterPFAttrTypeNum          1024  /* "MasterPF" */
#define  kMasterNetSpecAttrTypeNum     1025  /* "MasterNetSpec" */
#define  kServersOfAttrTypeNum         1026  /* "Servers Of" */
#define  kParentCIDAttrTypeNum         1027  /* "Parent CID" */
#define  kNetworkSpecAttrTypeNum       1028  /* "NetworkSpec" */
#define  kLocationAttrTypeNum          1029  /* "Location" */
#define  kTimeSvrTypeAttrTypeNum       1030  /* "TimeServer Type" */
#define  kUpdateTimerAttrTypeNum       1031  /* "Update Timer" */
#define  kShadowsOfAttrTypeNum         1032  /* "Shadows Of" */
#define  kShadowServerAttrTypeNum      1033  /* "Shadow Server" */
#define  kTBSetupAttrTypeNum           1034  /* "TB Setup" */
#define  kMailSetupAttrTypeNum         1035  /* "Mail Setup" */
#define  kSlotIDAttrTypeNum            1036  /* "SlotID" */
#define  kGatewayFileIDAttrTypeNum     1037  /* "Gateway FileID" */
#define  kMailServiceAttrTypeNum       1038  /* "Mail Service" */
#define  kStdSlotInfoAttrTypeNum       1039  /* "Std Slot Info" */
#define  kAssoDirectoryAttrTypeNum     1040  /* "Asso. Catalog" */
#define  kDirectoryAttrTypeNum         1041  /* "Catalog" */
#define  kDirectoriesAttrTypeNum       1042  /* "Catalogs" */
#define  kSFlagsAttrTypeNum            1043  /* "SFlags" */
#define  kLocalNameAttrTypeNum         1044  /* "Local Name" */
#define  kLocalKeyAttrTypeNum          1045  /* "Local Key" */
#define  kDirUserRIDAttrTypeNum        1046  /* "Dir User RID" */
#define  kDirUserKeyAttrTypeNum        1047  /* "Dir User Key" */
#define  kDirNativeNameAttrTypeNum     1048  /* "Dir Native Name" */
#define  kCommentAttrTypeNum           1049  /* "Comment" */
#define  kRealNameAttrTypeNum          1050  /* "Real Name" */
#define  kPrivateDataAttrTypeNum       1051  /* "Private Data" */
#define  kDirTypeAttrTypeNum           1052  /* "Catalog Type" */
#define  kDSAMFileAliasAttrTypeNum     1053  /* "CSAM File Alias" */
#define  kCanAddressToAttrTypeNum      1054  /* "Can Address To" */
#define  kDiscriminatorAttrTypeNum     1055  /* "Discriminator" */
#define  kAliasAttrTypeNum             1056  /* "Alias" */
#define  kParentMSAMAttrTypeNum        1057  /* "Parent MSAM" */
#define  kParentDSAMAttrTypeNum        1058  /* "Parent CSAM" */
#define  kSlotAttrTypeNum              1059  /* "Slot" */
```

```
#define  kAssoMailServiceAttrTypeNum   1060  /* "Asso. Mail Service" */
#define  kFakeAttrTypeNum              1061  /* "Fake" */
#define  kInheritSysAdminAttrTypeNum   1062  /* "Inherit SysAdministrators"*/
#define  kPreferredPDAttrTypeNum       1063  /* "Preferred PD" */
#define  kLastLoginAttrTypeNum         1064  /* "Last Login" */
#define  kMailerAOMStateAttrTypeNum    1065  /* "Mailer AOM State" */
#define  kMailerSendOptionsAttrTypeNum 1066  /* "Mailer Send Options" */
#define  kJoinedAttrTypeNum            1067  /* "Joined" */
#define  kUnconfiguredAttrTypeNum      1068  /* "Unconfigured" */
#define  kVersionAttrTypeNum           1069  /* "Version" */
#define  kLocationNamesAttrTypeNum     1070  /* "Location Names" */
#define  kActiveAttrTypeNum            1071  /* "Active" */
#define  kDeleteRequestedAttrTypeNum   1072  /* "Delete Requested" */
#define  kGatewayTypeAttrTypeNum       1073  /* "Gateway Type" */

#define  kFirstOCEAttrTypeNum kMemberAttrTypeNum/* first standard OCE
                                                   attribute type */

#define  kLastOCEAttrTypeNum kGatewayTypeAttrTypeNum/* last standard OCE
                                                       attribute type */

#define  kNumOCEAttrTypes (kLastOCEAttrTypeNum - kFirstOCEAttrTypeNum + 1)
                                                  /* the total number of
                                                   attributes */

/* OCE String Types */

typedef short CharacterSet;            /* script code info */

struct RString                         /* RString */
{
   RStringHeader
   Byte  body[kRStringMaxBytes];
};

typedef struct RString RString;
typedef RString *RStringPtr, **RStringHandle;

struct RString64                       /* RString64 */
{
   RStringHeader
   Byte     body[kRString64Size];
};

typedef struct RString64 RString64;
```

```
struct RString32                        /* RString32 */
{
   RStringHeader
   Byte     body[kRString32Size];
};

typedef struct RString32 RString32;

struct ProtoRString                     /* ProtoRString */
{
   RStringHeader
   /* The body for the ProtoRstring should be defined here */
};

typedef struct ProtoRString ProtoRString;
typedef ProtoRString *ProtoRString;

struct DirectoryName                    /* DirectoryName */
{
   RStringHeader
   Byte     body[kDirectoryNameMaxBytes];
};

typedef struct DirectoryName DirectoryName;
typedef DirectoryName *DirectoryNamePtr;

struct NetworkSpec                      /* NetworkSpec */
{
   RStringHeader
   Byte  body[kNetworkSpecMaxBytes];
};

typedef struct NetworkSpec NetworkSpec;
typedef NetworkSpec *NetworkSpecPtr;

typedef unsigned short RStringKind;

/* RecordID Types */
struct CreationID
{
   unsigned long  source;  /* private to a catalog.*/
   unsigned long  seq;     /* private to a catalog*/
};

typedef struct CreationID CreationID;
```

```
typedef CreationID AttributeCreationID;

struct LocalRecordID
{
   CreationID  cid;              /* creation ID of the record */
   RStringPtr  recordName;       /* name of the record */
   RStringPtr  recordType;       /* type of record */
};

typedef struct LocalRecordID LocalRecordID;
typedef LocalRecordID *LocalRecordIDPtr;

struct PackedPathName
{
   unsigned short dataLength;    /* number of bytes in data field */
   Byte           data[kPathNameMaxBytes - sizeof (unsigned short)];
};

typedef struct PackedPathName PackedPathName;
typedef PackedPathName *PackedPathNamePtr;

struct ProtoPackedPathName {

   unsigned short dataLength;

/* Followed by data */

};

typedef struct ProtoPackedPathName ProtoPackedPathName;
typedef ProtoPackedPathName *ProtoPackedPathNamePtr;

struct DirDiscriminator {
   OCEDirectoryKind  signature;     /* private to catalog */
   unsigned long     misc;          /* private to catalog */
};

typedef struct DirDiscriminator DirDiscriminator;

typedef unsigned long   DNodeNum;

struct RLI {
   DirectoryNamePtr     directoryName;
   DirDiscriminator     discriminator;
```

```
   DNodeNum             dNodeNumber;
   PackedPathNamePtr    path;
};

typedef struct RLI RLI;
typedef RLI *RLIPtr;

struct PackedRLI {
   dataLength;
   Byte           data[kRLIMaxBytes];  /* packed record
                                          location info */
};

typedef struct PackedRLI PackedRLI;
typedef PackedRLI *PackedPLIPtr;

struct ProtoPackedRLI {
   dataLength
/* Followed by data */
};

typedef struct ProtoPackedRLI ProtoPackedRLI;
typedef ProtoPackedRLI *ProtoPackedRLIPtr;

struct RecordID {
   PackedRLIPtr      rli;     /* identifies record's catalog
                                 and dNode */
   LocalRecordID     local;   /* identifies record within
                                 its dNode */
};

typedef struct RecordID RecordID;
typedef RecordID *RecordIDPtr;

struct PackedRecordID {
   dataLength;                          /* length of data field */
   Byte  data[kPackedRecordIDMaxBytes];/* packed record ID */
};

typedef struct PackedRecordID PackedRecordID;
typedef PackedRecordID *PackedRecordIDPtr;
```

```
struct ShortRecordID
{
   PackedRLIPtr rli;
   CreationID cid;
};

typedef struct ShortRecordID ShortRecordID;
typedef ShortRecordID *ShortRecordIDPtr;

/* DSSpec Structures */
struct DSSpec {
   RecordID    *entitySpecifier;
   OSType      extensionType;
   unsigned    short extensionSize;
   Ptr         extensionValue;
};

typedef struct DSSpec DSSpec;
typedef DSSpec *DSSpecPtr;

struct PackedDSSpec {
   dataLength
   Byte        data[kPackedDSSpecMaxBytes];
};

typedef struct PackedDSSpec PackedDSSpec;
typedef PackedDSSpec *PackedDSSpecPtr;

struct ProtoPackedDSSpec {
   dataLength
/* Followed by data */
};

typedef struct ProtoPackedDSSpec ProtoPackedDSSpec;
typedef ProtoPackedDSSpec *ProtoPackedDSSpecPtr;

/* Attribute Structures */
struct AttributeType
{
   RStringHeader
   Byte  body[kAttributeTypeMaxBytes];
};

typedef struct AttributeType AttributeType;
typedef AttributeType *AttributeTypePtr;
```

```
struct AttributeValue {
   AttributeTag   tag;        /* format of attribute value */
   unsigned long  dataLength; /* # of bytes in attribute value */
   Ptr            bytes;      /* points to attribute value data */
};

typedef struct AttributeValue AttributeValue;
typedef AttributeValue *AttributeValuePtr;

typedef CreationID   AttributeCreationID;

struct Attribute {
   AttributeType        attributeType; /* type of the attribute */
   AttributeCreationID  cid;           /* the creationID of the
                                          attribute */
   AttributeValue       value;         /* the attribute value */
};

typedef struct Attribute Attribute;
typedef Attribute *AttributePtr;

typedef DescType AttributeTag;/* same type used in AppleEvents */

/* recordType index */

typedef unsigned short OCERecordTypeIndex;

/* AttributeType index */

typedef unsigned short OCEAttributeTypeIndex;

/* OCE Catalog Types */

typedef unsigned long OCEDirectoryKind;

/* OCE Catalog Node Types */

typedef unsigned long DirNodeKind;
```

AOCE Utility Functions

AOCE String Functions

```
pascal OSErr OCECopyRString
                           (const RString *str1, RString *str2, unsigned
                            short str2Length);
```

```
pascal void OCECToRString   (const char *cStr, CharacterSet charSet,RString
                             *rStr,unsigned short rStrLength);
pascal void OCEPToRString   (ConstStr255Param pStr,CharacterSet charSet,
                             RString *rStr,unsigned short rStrLength);
pascal StringPtr OCERToPString
                            (const RString *rStr);
pascal short OCERelRString  (const void *str1, const void *str2,RStringKind
                             kind);
pascal Boolean OCEEqualRString
                            (const void *str1,const void *str2,RStringKind
                             kind);
pascal Boolean OCEValidRString
                            (const void *str,RStringKind kind);
```

Creation Identifier Functions

```
pascal Boolean OCEEqsrualCreationID
                            (const CreationID *cid1,
                             const CreationID *cid2);
pascal void OCECopyCreationID
                            (const CreationID *cid1,CreationID *const cid2);
pascal const CreationID     *OCENullCID(void);
pascal const CreationID     *OCEPathFinderCID(void);
pascal void OCESetCreationIDtoNull
                            (CreationID *const cid);
```

Packed Pathname Functions

```
pascal OSErr OCECopyPackedPathName
                            (const PackedPathName *path1,PackedPathName
                             *path2,unsigned short path2Length);
pascal Boolean OCEIsNullPackedPathName
                            (const PackedPathName *path);
pascal unsigned short OCEPackedPathNameSize
                            (const RStringPtr parts[], const unsigned short
                             nParts);
pascal unsigned short OCEDNodeNameCount
                            (const PackedPathName *path);
pascal unsigned short OCEUnpackPathName
                            (const PackedPathName *path, RString *const
                             parts[], const unsigned short nParts);
```

```
pascal OSErr OCEPackPathName
                            (const RStringPtr parts[],const unsigned short
                             nParts,PackedPathName *path,unsigned short
                             pathLength);
pascal Boolean OCEEqualPackedPathName
                            (const PackedPathName *path1, const
                             PackedPathName *path2);
pascal Boolean OCEValidPackedPathName
                            (const PackedPathName *path);
```

Catalog Discriminator Functions

```
pascal void OCECopyDirDiscriminator
                            (const DirDiscriminator *disc1,
                             DirDiscriminator *const disc2);
pascal Boolean OCEEqualDirDiscriminator
                            (const DirDiscriminator *disc1, const
                             DirDiscriminator *disc2);
```

Record Location Information Functions

```
pascal void OCENewRLI       (RLI *newRLI, const DirectoryName *dirName,
                             DirDiscriminator *discriminator,const DNodeNum
                             dNodeNumber,const PackedPathName *path);
pascal void OCEDuplicateRLI (const RLI *rli1, RLI *rli2);
pascal OSErr OCECopyRLI     (const RLI *rli1, RLI *rli2);
pascal Boolean OCEEqualRLI  (const RLI *rli1, const RLI *rli2);
pascal Boolean OCEValidRLI  (const RLI *theRLI);
pascal OSErr OCECopyPackedRLI
                            (const PackedRLI *prli1, PackedRLI
                             *prli2,unsigned short prli2Length);
pascal unsigned short OCEPackedRLISize
                            (const RLI *theRLI);
pascal OSErr OCEPackRLI     (const RLI *theRLI, PackedRLI *prli, unsigned
                             short prliLength);
pascal void OCEUnpackRLI    (const PackedRLI *prli, RLI *theRLI);
pascal unsigned short OCEPackedRLIPartsSize
                            (const DirectoryName *dirName, const RStringPtr
                             parts[], const unsigned short nParts);
```

```
pascal OSErr OCEPackRLIParts
                            (const DirectoryName *dirName, const
                             DirDiscriminator *discriminator, const
                             DNodeNum dNodeNumber, const RStringPtr
                             parts[], const unsigned short nParts,
                             PackedRLI *prli, unsigned short prliLength);
pascal Boolean OCEEqualPackedRLI
                            (const PackedRLI *prli1, const PackedRLI
                             *prli2);
pascal Boolean OCEValidPackedRLI
                            (const PackedRLI *prli);
pascal AliasPtr OCEExtractAlias
                            (const PackedRLI *prli);
pascal const PackedRLI      * OCEGetDirectoryRootPackedRLI (void)
```

Local Record Identifier Functions

```
pascal void OCENewLocalRecordID
                            (const RString *recordName, const RString
                             *recordType, const CreationID *cid,
                             LocalRecordID *lRID);
pascal OSErr OCECopyLocalRecordID
                            (const LocalRecordID *lRID1, LocalRecordID
                             *lRID2);
pascal Boolean OCEEqualLocalRecordID
                            (const LocalRecordID *lRID1, const
                             LocalRecordID *lRID2);
```

Short Record Identifier Functions

```
pascal void OCENewShortRecordID
                            (const PackedRLI *theRLI, const CreationID
                             *cid, ShortRecordIDPtr *sRID);
pascal OSErr OCECopyShortRecordID
                            (const ShortRecordID *sRID1,ShortRecordID
                             *sRID2);
pascal Boolean OCEEqualShortRecordID
                            (const ShortRecordID *sRID1,const ShortRecordID
                             *sRID2);
```

Record Identifier Functions

```
pascal RString *OCEGetIndRecordType
                            (const OCERecordTypeIndex stringIndex);
```

```
pascal void OCENewRecordID  (const PackedRLI *theRLI, const LocalRecordID
                             *lRID, RecordID *rid);
pascal OSErr OCECopyRecordID
                            (const RecordID *rid1,const RecordID *rid2);
pascal Boolean OCEEqualRecordID
                            (const RecordID *rid1,const RecordID *rid2);
```

Packed Record Identifier Functions

```
pascal OSErr OCECopyPackedRecordID
                            (const PackedRecordID *pRID1, const
                             PackedRecordID *pRID2, unsigned short
                             pRID2length);
pascal unsigned short OCEPackedRecordIDSize
                            (const RecordID *rid);
pascal OSErr OCEPackRecordID
                            (const RecordID *rid, PackedRecordID *pRID,
                             unsigned short packedRecordIDlength);
pascal void OCEUnpackRecordID
                            (const PackedRecordID *pRID, RecordID *rid);
pascal Boolean OCEEqualPackedRecordID
                            (const PackedRecordID *pRID1, const
                             PackedRecordID *pRID2);
pascal Boolean OCEValidPackedRecordID
                            (const PackedRecordID *pRID);
```

Attribute Type Functions

```
pascal AttributeType        *OCEGetIndAttributeType(const
                             OCEAttributeTypeIndex stringIndex);
```

Catalog Services Specification Functions

```
pascal OSErr OCECopyPackedDSSpec
                            (const PackedDSSpec *pdss1, const PackedDSSpec
                             *pdss2, unsigned short pdss2Length);
pascal unsigned short OCEPackedDSSpecSize
                            (const DSSpec *dss);
pascal OSErr OCEPackDSSpec  (const DSSpec *dss, PackedDSSpec *pdss,
                             unsigned short pdssLength);
pascal void OCEUnpackDSSpec (const PackedDSSpec *pdss, DSSpec *dss,
                             RecordID *rid);
pascal Boolean OCEEqualDSSpec
                            (const DSSpec *pdss1, const DSSpec *pdss2);
```

```
pascal Boolean OCEEqualPackedDSSpec
                            (const PackedDSSpec *pdss1, const PackedDSSpec
                             *pdss2);
pascal Boolean OCEValidPackedDSSpec
                            (const PackedDSSpec *pdss);
pascal OSType OCEGetDSSpecInfo
                            (const DSSpec *spec);
pascal OSType OCEGetExtensionType
                            (const PackedDSSpec *pdss);
pascal OSErr OCEStreamPackedDSSpec
                            (const DSSpec *dss,MyDSSpecStreamer stream,
                             long userData, unsigned long *actualCount);
```

Application-Defined Functions

```
typedef pascal OSErr        (*MyDSSpecStreamer)(void *buffer, unsigned long
                             count, Boolean eof, long userData);
```

Pascal Summary

Constants

```
CONST

{ OCE String Constants }
   kRString32Size             = 32;    { max size of RString32 }
   kRString64Size             = 64;    { max size of RString64 }
   kNetworkSpecMaxBytes       = 32;    { max size of NetworkSpec }
   kPathNameMaxBytes          = 1024;  { max size of PackedPathName }
   kDirectoryNameMaxBytes     = 32;    { max size of DirectoryName }
   kAttributeTypeMaxBytes     = 32;    { max size of AttributeType }
   kAttrValueMaxBytes         = 65536; { max size of any attribute value }
   kRStringMaxBytes           = 256;   { max bytes in recordName,recordType }
   kRStringMaxChars           = 128;   { max chars in recordName,recordType }

   kMinPackedRStringLength = sizeof(ProtoRString);

{ values of RStringKind }
   kOCEDirName             = 0;
   kOCERecordOrDNodeName   = 1;
   kOCERecordType          = 2;
   kOCENetworkSpec         = 3;
```

```
   kOCEAttrType            = 4;
   kOCEGenericSensitive    = 5;
   kOCEGenericInsensitive  = 6;

{ values of OCEDirectoryKind }
   kDirAllKinds            = 0;
   kDirADAPKind            = 'adap';
   kDirPersonalDirectoryKind
                           = 'pdir';
   kDirDSAMKind            = 'dsam';

{ Catalog Node Constants }
   kNULLDNodeNumber          = 0;        { none specified }
   kRootDNodeNumber          = 2;        { the root of the tree }

{ Values returned by OCEGetDSSpecInfo() }
   kOCEInvalidDSSpec       = '????',   { could not be determined }
   kOCEDirsRootDSSpec      = 'root',   { root of all catalogs }
   kOCEDirectoryDSSpec     = 'dire',   { catalog }
   kOCEDNodeDSSpec         = 'dnod',   { Dnode }
   kOCERecordDSSpec        = 'reco',   { record }
   kOCEentnDSSpec          = 'entn',   { extensionType is 'entn' }
   kOCENOTentnDSSpec       = 'not '    { extensionType is not 'entn' }

{ AttributeTag Values }
   typeRString             = 'rstr',   { attribute value is an RString }
   typePackedDSSpec        = 'dspc',   { attribute value is a DSSpec }
   typeBinary              = 'bnry'    { attribute value is a sequence
                                         of bytes }

{ Cluster info }
   kcanContainRecordsBit,  = 0{ a cluster }
   kForeignNodeBit         = 1{ a foreign catalog }

{ values of DirNodeKind }
   kcanContainRecords      = $00000001;{<<kcanContainRecordsBit}
   kForeignNode            = $00000002;{<<kForeignNodeBit}

{ RLI Constants }

   kRLIMaxBytes      = (sizeof (RString) + sizeof (DirDiscriminator) +
                        sizeof (DNodeNum) + kPathNameMaxBytes);
```

```
   kMinPackedRLISize  = (sizeof (ProtoPackedRLI) +
                           sizeof (DirDiscriminator) + sizeof (DNodeNum) +
                           kMinPackedRStringLength +
                           sizeof (ProtoPackedPathName));

{ RecordID Constants }
   kPackedRecordIDMaxBytes = kPathNameMaxBytes + sizeof(DNodeNum) +
      sizeof(DirDiscriminator) + sizeof(CreationID) + (3*sizeof(RString));

{ DSSpec Constants }

   kPackedDSSpecMaxBytes = (sizeof (PackedRecordID) + sizeof (OSType) +
                           sizeof (INTEGER));

   { Indices for the standard definitions for standard record types }

   kUserRecTypeNum                        = 1;  { "User" }
   kGroupRecTypeNum                       = 2;  { "Group" }
   kMnMRecTypeNum                         = 3;  { "AppleMail™ M&M" }
   kMnMForwarderRecTypeNum                = 4;  { "AppleMail™ Fwdr" }
   kNetworkSpecRecTypeNum                 = 5;  { "NetworkSpec" }
   kADAPServerRecTypeNum                  = 6;  { "PowerShare Server" }
   kADAPDNodeRecTypeNum                   = 7;  { "PowerShare DNode" }
   kADAPDNodeRepRecTypeNum                = 8;  { "PowerShare DNode Rep" }
   kServerSetupRecTypeNum                 = 9;  { "Server Setup" }
   kDirectoryRecTypeNum                   = 10; { "Catalog" }
   kDNodeRecTypeNum                       = 11; { "DNode" }
   kSetupRecTypeNum                       = 12; { "Setup" }
   kMSAMRecTypeNum                        = 13; { "MSAM" }
   kDSAMRecTypeNum                        = 14; { "CSAM" }
   kAttributeValueRecTypeNum              = 15; { "Attribute Value" }
   kBusinessCardRecTypeNum                = 16; { "Business Card" }
   kMailServiceRecTypeNum                 = 17; { "Mail Service" }
   kCombinedRecTypeNum                    = 18; { "Combined" }
   kOtherServiceRecTypeNum                = 19; { "Other Service" }
   kAFPServiceRecTypeNum                  = 20; { "Other Service afps" }

   kFirstOCERecTypeNum = kUserRecTypeNum;       { first standard OCE record
                                                  type }

   kLastOCERecTypeNum = kAFPServiceRecTypeNum;  { last standard OCE record
                                                  type }

   kNumOCERecTypes = (kLastOCERecTypeNum - kFirstOCERecTypeNum + 1);
```

```
{ Indices for the standard definitions for standard attribute types
(OCEAttributeTypeIndex): }


   kMemberAttrTypeNum                    = 1001;{ "Member" }
   kAdminsAttrTypeNum                    = 1002;{ "Administrators" }
   kMailSlotsAttrTypeNum                 = 1003;{ "mailslots" }
   kPrefMailAttrTypeNum                  = 1004;{ "pref mailslot" }
   kAddressAttrTypeNum                   = 1005;{ "Address" }
   kPictureAttrTypeNum                   = 1006;{ "Picture" }
   kAuthKeyAttrTypeNum                   = 1007;{ "auth key" }
   kTelephoneAttrTypeNum                 = 1008;{ "Telephone" }
   kNBPNameAttrTypeNum                   = 1009;{ "NBP Name" }
   kQMappingAttrTypeNum                  = 1010;{ "ForwarderQMap" }
   kDialupSlotAttrTypeNum                = 1011;{ "DialupSlotInfo" }
   kHomeNetAttrTypeNum                   = 1012;{ "Home Internet" }
   kCoResAttrTypeNum                     = 1013;{ "Co-resident M&M" }
   kFwdrLocalAttrTypeNum                 = 1014;{ "FwdrLocalRecord" }
   kConnectAttrTypeNum                   = 1015;{ "Connected To" }
   kForeignAttrTypeNum                   = 1016;{ "Foreign RLIs" }
   kOwnersAttrTypeNum                    = 1017;{ "Owners" }
   kReadListAttrTypeNum                  = 1018;{ "ReadList" }
   kWriteListAttrTypeNum                 = 1019;{ "WriteList" }
   kDescriptorAttrTypeNum                = 1020;{ "Descriptor" }
   kCertificateAttrTypeNum               = 1021;{ "Certificate" }
   kMsgQsAttrTypeNum                     = 1022;{ "MessageQs" }
   kPrefMsgQAttrTypeNum                  = 1023;{ "PrefMessageQ" }
   kMasterPFAttrTypeNum                  = 1024;{ "MasterPF" }
   kMasterNetSpecAttrTypeNum             = 1025;{ "MasterNetSpec" }
   kServersOfAttrTypeNum                 = 1026;{ "Servers Of" }
   kParentCIDAttrTypeNum                 = 1027;{ "Parent CID" }
   kNetworkSpecAttrTypeNum               = 1028;{ "NetworkSpec" }
   kLocationAttrTypeNum                  = 1029;{ "Location" }
   kTimeSvrTypeAttrTypeNum               = 1030;{ "TimeServer Type" }
   kUpdateTimerAttrTypeNum               = 1031;{ "Update Timer" }
   kShadowsOfAttrTypeNum                 = 1032;{ "Shadows Of" }
   kShadowServerAttrTypeNum              = 1033;{ "Shadow Server" }
   kTBSetupAttrTypeNum                   = 1034;{ "TB Setup" }
   kMailSetupAttrTypeNum                 = 1035;{ "Mail Setup" }
   kSlotIDAttrTypeNum                    = 1036;{ "SlotID" }
   kGatewayFileIDAttrTypeNum             = 1037;{ "Gateway FileID" }
   kMailServiceAttrTypeNum               = 1038;{ "Mail Service" }
   kStdSlotInfoAttrTypeNum               = 1039;{ "Std Slot Info" }
   kAssoDirectoryAttrTypeNum             = 1040;{ "Asso. Catalog" }
```

```
kDirectoryAttrTypeNum                  = 1041;{ "Catalog" }
kDirectoriesAttrTypeNum                = 1042;{ "Catalogs" }
kSFlagsAttrTypeNum                     = 1043;{ "SFlags" }
kLocalNameAttrTypeNum                  = 1044;{ "Local Name" }
kLocalKeyAttrTypeNum                   = 1045;{ "Local Key" }
kDirUserRIDAttrTypeNum                 = 1046;{ "Dir User RID" }
kDirUserKeyAttrTypeNum                 = 1047;{ "Dir User Key" }
kDirNativeNameAttrTypeNum              = 1048;{ "Dir Native Name" }
kCommentAttrTypeNum                    = 1049;{ "Comment" }
kRealNameAttrTypeNum                   = 1050;{ "Real Name" }
kPrivateDataAttrTypeNum                = 1051;{ "Private Data" }
kDirTypeAttrTypeNum                    = 1052;{ "Catalog Type" }
kDSAMFileAliasAttrTypeNum              = 1053;{ "CSAM File Alias" }
kCanAddressToAttrTypeNum               = 1054;{ "Can Address To" }
kDiscriminatorAttrTypeNum              = 1055;{ "Discriminator" }
kAliasAttrTypeNum                      = 1056;{ "Alias" }
kParentMSAMAttrTypeNum                 = 1057;{ "Parent MSAM" }
kParentDSAMAttrTypeNum                 = 1058;{ "Parent CSAM" }
kSlotAttrTypeNum                       = 1059;{ "Slot" }
kAssoMailServiceAttrTypeNum            = 1060;{ "Asso. Mail Service" }
kFakeAttrTypeNum                       = 1061;{ "Fake" }
kInheritSysAdminAttrTypeNum            = 1062;{ "Inherit System
                                                 Administrators" }
kPreferredPDAttrTypeNum                = 1063;{ "Preferred PD" }
kLastLoginAttrTypeNum                  = 1064;{ "Last Login" }
kMailerAOMStateAttrTypeNum             = 1065;{ "Mailer AOM State" }
kMailerSendOptionsAttrTypeNum          = 1066;{ "Mailer Send Options" }
kJoinedAttrTypeNum                     = 1067;{ "Joined" }
kUnconfiguredAttrTypeNum               = 1068;{ "Unconfigured" }
kVersionAttrTypeNum                    = 1069;{ "Version" }
kLocationNamesAttrTypeNum              = 1070;{ "Location Names" }
kActiveAttrTypeNum                     = 1071;{ "Active" }
kDeleteRequestedAttrTypeNum            = 1072;{ "Delete Requested" }
kGatewayTypeAttrTypeNum                = 1073;{ "Gateway Type" }

kFirstOCEAttrTypeNum = kMemberAttrTypeNum;{ first standard OCE attr type }

kLastOCEAttrTypeNum  = kGatewayTypeAttrTypeNum;{ last standard OCE
                                                 attr type }

kNumOCEAttrTypes     = (kLastOCEAttrTypeNum - kFirstOCEAttrTypeNum + 1);
```

Data Types

```
TYPE

{ OCE String Types }

   {RStringHeader}
   RStringHeader = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
   END;

   { RString }
   RString = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      body: PACKED ARRAY[1..kRStringMaxBytes] OF Byte;
   END;

   { ProtoRString }
   ProtoRString = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      { Followed by body }
   END;

   RStringPtr = ^RString;

   RStringHandle = ^RStringPtr;

   ProtoRStringPtr = ^ProtoRString;

   {RString64}
   RString64 = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      body: PACKED ARRAY[1..kRString64Size] OF Byte;
   END;

   {RString32}
   RString32 = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      body: PACKED ARRAY[1..kRString32Size] OF Byte;
   END;
```

```
   Rstring32Ptr = ^Rstring32;

struct DirectoryName                    /* DirectoryName */
{
   RStringHeader
   Byte     body[kDirectoryNameMaxBytes];
};

   {NetworkSpec}
   NetworkSpec = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      body: PACKED ARRAY[1..kNetworkSpecMaxBytes] OF Byte;
   END;

   NetworkSpecPtr = ^NetworkSpec;

   RStringKind = INTEGER;

{ RecordID Types }

   {CreationID}
   CreationID = RECORD
      source: LONGINT;
      seq: LONGINT;
   END;

   AttributeCreationID = CreationID;

   CreationIDPtr = ^CreationID;

      {PackedPathName}
   PackedPathName = RECORD
      dataLength: INTEGER;
      data: PACKED ARRAY[1..kPathNameMaxBytes - sizeof(INTEGER)] OF Byte;
   END;

   {ProtoPackedPathName}
   ProtoPackedPathName = RECORD
      dataLength: INTEGER;
      { Followed by data }
   END;

   PackedPathNamePtr = ^PackedPathName;

   ProtoPackedPathNamePtr = ^ProtoPackedPathName;
```

```
{DirDiscriminator}
DirDiscriminator = RECORD
   signature: OCEDirectoryKind;
   misc: LONGINT;
END;

{ Catalog node number }
DNodeNum = LONGINT;

{ RLI }
RLI = RECORD
   directoryName: DirectoryNamePtr;
   discriminator: DirDiscriminator;
   dNodeNumber: DNodeNum;
   path: PackedPathNamePtr;
END;

RLIPtr = ^RLI;

{ PackedRLIHeader }
PackedRLIHeader = RECORD
   dataLength: INTEGER;
END;

{ PackedRLI }
PackedRLI = RECORD
   dataLength: INTEGER;
   data: PACKED ARRAY[1..kRLIMaxBytes] OF Byte;
END;

{ ProtoPackedRLI }
ProtoPackedRLI = RECORD
   dataLength: INTEGER;
   { Followed by data }
END;

PackedRLIPtr = ^PackedRLI;

ProtoPackedRLIPtr = ^ProtoPackedRLI;

{ LocalRecordID }
LocalRecordID = RECORD
   cid: CreationID;
```

```
      recordName: RStringPtr;
      recordType: RStringPtr;
   END;

   LocalRecordIDPtr = ^LocalRecordID;

   { ShortRecordID }
   ShortRecordID = RECORD
      rli: PackedRLIPtr;
      cid: CreationID;
   END;

   ShortRecordIDPtr = ^ShortRecordID;

   { RecordID }
   RecordID = RECORD
      rli: PackedRLIPtr;
      local: LocalRecordID;
   END;

   RecordIDPtr = ^RecordID;

   { PackedRecordIDHeader }
   PackedRecordIDHeader = RECORD
      dataLength: INTEGER;
   END;

   { PackedRecordID }
   PackedRecordID = RECORD
      dataLength: INTEGER;
      data: PACKED ARRAY[1..kPackedRecordIDMaxBytes] OF Byte;
   END;

   { ProtoPackedRecordID }
   ProtoPackedRecordID = RECORD
      dataLength: INTEGER;
      { Followed by data }
   END;

   PackedRecordIDPtr = ^PackedRecordID;

   ProtoPackedRecordIDPtr = ^ProtoPackedRecordID;

{ DSSpec Structures }
```

```
   { DSSpec }
   DSSpec = RECORD
      entitySpecifier: ^RecordID;
      extensionType: OSType;
      extensionSize: INTEGER;
      extensionValue: Ptr;
   END;

   DSSpecPtr = ^DSSpec;

   { PackedDSSpecHeader }
   PackedDSSpecHeader = RECORD
      dataLength: INTEGER;
   END;

   { PackedDSSpec }
   PackedDSSpec = RECORD
      dataLength: INTEGER;
      data: PACKED ARRAY[1..kPackedDSSpecMaxBytes] OF Byte;
   END;

   ProtoPackedDSSpec = RECORD
      dataLength: INTEGER;
      { Followed by data }
   END;

   PackedDSSpecPtr = ^PackedDSSpec;

   PackedDSSpecHandle = ^PackedDSSpecPtr;

   ProtoPackedDSSpecPtr = ^ProtoPackedDSSpec;

{ Attribute Structures }
   AttributeType = RECORD
      charSet: CharacterSet;
      dataLength: INTEGER;
      body: PACKED ARRAY[1..kAttributeTypeMaxBytes] OF Byte;
   END;

   AttributeTypePtr = ^AttributeType;

   { AttributeValue }
   AttributeValue = RECORD
      tag: AttributeTag;
```

```
      dataLength: LONGINT;
      bytes: Ptr;
   END;

   AttributeValuePtr = ^AttributeValue;

   AttributeTag = DescType;

   { Attribute }
   Attribute = RECORD
      attributeType: AttributeType;
      cid: AttributeCreationID;
      value: AttributeValue;
   END;

   AttributePtr = ^Attribute;

{ recordType index }
   OCERecordTypeIndex = INTEGER;

{ AttributeType index }
   OCEAttributeTypeIndex = INTEGER;

{ OCE Catalog Types }
   OCEDirectoryKind = LONGINT;

{ OCE Catalog Node Types }
   DirNodeKind = LONGINT;

{ MyDSSpecStreamer callback routine }
   MyDSSpecStreamer = ProcPtr;
```

AOCE Utility Functions

AOCE String Functions

```
FUNCTION OCECopyRString     (str1: RStringPtr; str2: RStringPtr;
                             str2Length: INTEGER): OSErr;
PROCEDURE OCECToRString     (cStr: Ptr; charSet: CharacterSet; rStr:
                             RStringPtr; rStrLength: INTEGER);
PROCEDURE OCEPToRString     (pStr: Str255; charSet: CharacterSet; rStr:
                             RStringPtr; rStrLength: INTEGER);
FUNCTION OCERToPString      (rStr: RStringPtr): StringPtr; INLINE $303C,
                             kOCERToPString, $AA5C;
```

```
FUNCTION OCERelRString      (str1: UNIV Ptr; str2: UNIV Ptr; kind:
                             RStringKind): INTEGER;
FUNCTION OCEEqualRString    (str1: UNIV Ptr; str2: UNIV Ptr; kind:
                             RStringKind): BOOLEAN;
FUNCTION OCEValidRString    (str: UNIV Ptr; kind: RStringKind): BOOLEAN;
```

Creation Identifier Functions

```
FUNCTION OCEEqualCreationID (cid1: CreationID; cid2: CreationID): BOOLEAN;
PROCEDURE OCECopyCreationID (cid1: CreationID; VAR cid2: CreationID);
FUNCTION OCENullCID: CreationIDPtr;
FUNCTION OCEPathFinderCID: CreationIDPtr;
PROCEDURE OCESetCreationIDtoNull
                            (VAR cid: CreationID);
```

Packed pathname Functions

```
FUNCTION OCECopyPackedPathName
                            (path1: PackedPathNamePtr; path2:
                             PackedPathNamePtr; path2Length: INTEGER):
                             OSErr;
FUNCTION OCEIsNullPackedPathName
                            (path: PackedPathNamePtr): BOOLEAN;
FUNCTION OCEPackedPathNameSize
                            (VAR parts: RStringPtr; nParts: INTEGER):
                             INTEGER;
FUNCTION OCEDNodeNameCount  (path: PackedPathNamePtr): INTEGER;
FUNCTION OCEUnpackPathName  (path: PackedPathNamePtr; VAR parts:
                             RStringPtr; nParts: INTEGER): INTEGER;
FUNCTION OCEPackPathName    (VAR parts: RStringPtr; nParts: INTEGER; path:
                             PackedPathNamePtr; pathLength: INTEGER): OSErr;
FUNCTION OCEEqualPackedPathName
                            (path1: PackedPathNamePtr; path2:
                             PackedPathNamePtr): BOOLEAN;
FUNCTION OCEValidPackedPathName
                            (path: PackedPathNamePtr): BOOLEAN;
```

Catalog Discriminator Functions

```
PROCEDURE OCECopyDirDiscriminator
                            (disc1: DirDiscriminator; VAR disc2:
                             DirDiscriminator);
```

```
FUNCTION OCEEqualDirDiscriminator
                            (disc1: DirDiscriminator; disc2:
                             DirDiscriminator): BOOLEAN;
```

Record Location Information Functions

```
PROCEDURE OCENewRLI         (VAR newRLI: RLI; dirName: DirectoryName; VAR
                             discriminator: DirDiscriminator dNodeNumber:
                             DNodeNum; path: PackedPathName);
PROCEDURE OCEDuplicateRLI   (rli1: RLI; VAR rli2: RLI);
FUNCTION OCECopyRLI         (rli1: RLI; VAR rli2: RLI): OSErr;
FUNCTION OCEEqualRLI        (rli1: RLI; rli2: RLI): BOOLEAN;
FUNCTION OCEValidRLI        (theRLI: RLI): BOOLEAN;
FUNCTION OCECopyPackedRLI   (prli1: PackedRLIPtr; prli2: PackedRLIPtr;
                             prli2Length: INTEGER): OSErr;
FUNCTION OCEPackedRLISize   (theRLI: RLI): INTEGER;
FUNCTION OCEPackRLI         (theRLI: RLI; prli: PackedRLIPtr; prliLength:
                             INTEGER): OSErr;
PROCEDURE OCEUnpackRLI      (prli: PackedRLIPtr; VAR theRLI: RLI);
FUNCTION OCEPackedRLIPartsSize
                            (dirName: DirectoryNamePtr; VAR parts:
                             RStringPtr; nParts: INTEGER): INTEGER;
FUNCTION OCEPackRLIParts    (dirName: DirectoryNamePtr; discriminator:
                             DirDiscriminator; dNodeNumber: DNodeNum; VAR
                             parts: RStringPtr; nParts: INTEGER; prli:
                             PackedRLIPtr; prliLength: INTEGER): OSErr;
FUNCTION OCEEqualPackedRLI  (prli1: PackedRLIPtr; prli2: PackedRLIPtr):
                             BOOLEAN;
FUNCTION OCEValidPackedRLI  (prli: PackedRLIPtr): BOOLEAN;
FUNCTION OCEExtractAlias    (prli: PackedRLIPtr): AliasPtr;
FUNCTION OCEGetDirectoryRootPackedRLI
                            ():PackedRLIPtr;
```

Local Record Identifier Functions

```
PROCEDURE OCENewLocalRecordID
                            (recordName: RStringPtr; recordType:RStringPtr;
                             cid: CreationID; VAR lRID: LocalRecordID);
FUNCTION OCECopyLocalRecordID
                            (lRID1: LocalRecordID; VAR lRID2:
                             LocalRecordID): OSErr;
FUNCTION OCEEqualLocalRecordID
                            (lRID1: LocalRecordID; lRID2: LocalRecordID):
                             BOOLEAN;
```

Short Record Identifier Functions

```
PROCEDURE OCENewShortRecordID
                            (theRLI: PackedRLIPtr; cid: CreationID; sRID:
                             ShortRecordIDPtr);
FUNCTION OCECopyShortRecordID
                            (sRID1: ShortRecordID; VAR sRID2:
                             ShortRecordID): OSErr;
FUNCTION OCEEqualShortRecordID
                            (sRID1: ShortRecordID; sRID2: ShortRecordID):
                             BOOLEAN;
```

Record Identifier Functions

```
FUNCTION OCEGetIndRecordType
                            (STRINGIndex: OCERecordTypeIndex): RStringPtr;
PROCEDURE OCENewRecordID    (theRLI: PackedRLIPtr; lRID: LocalRecordID; VAR
                             rid: RecordID);
FUNCTION OCECopyRecordID    (rid1: RecordID; rid2: RecordID): OSErr;
FUNCTION OCEEqualRecordID   (rid1: RecordID; rid2: RecordID): BOOLEAN;
```

Packed Record Identifier Functions

```
FUNCTION OCECopyPackedRecordID
                            (pRID1: PackedRecordIDPtr; pRID2:
                             PackedRecordIDPtr; pRID2Length: INTEGER):
                             OSErr;
FUNCTION OCEPackedRecordIDSize
                            (rid: RecordID): INTEGER;
FUNCTION OCEPackRecordID    (rid: RecordID; VAR pRID: PackedRecordIDPtr;
                             packedRecordIDLength: INTEGER): OSErr;
PROCEDURE OCEUnpackRecordID (pRID: PackedRecordIDPtr; VAR rid: RecordID);
FUNCTION OCEEqualPackedRecordID
                            (pRID1: PackedRecordIDPtr; pRID2:
                             PackedRecordIDPtr): BOOLEAN;
FUNCTION OCEValidPackedRecordID
                            (pRID: PackedRecordIDPtr): BOOLEAN;
```

Attribute Type Functions

```
FUNCTION OCEGetIndAttributeType
                            (STRINGIndex: OCEAttributeTypeIndex):
                             AttributeTypePtr;
```

Catalog Services Specification Functions

```
FUNCTION OCECopyPackedDSSpec
                            (pdss1: PackedDSSpecPtr; pdss2:
                             PackedDSSpecPtr; pdss2Length: INTEGER): OSErr;
FUNCTION OCEPackedDSSpecSize
                            (dss: DSSpec): INTEGER;
FUNCTION OCEPackDSSpec      (dss: DSSpec; VAR pdss: PackedDSSpecPtr;
                             pdssLength: INTEGER): OSErr;
PROCEDURE OCEUnpackDSSpec   (pdss: PackedDSSpecPtr; VAR dss: DSSpec; VAR
                             rid: RecordID);
FUNCTION OCEEqualDSSpec     (pdss1: DSSpec; pdss2: DSSpec): BOOLEAN;
FUNCTION OCEEqualPackedDSSpec
                            (pdss1: PackedDSSpecPtr; pdss2:
                             PackedDSSpecPtr): BOOLEAN;
FUNCTION OCEValidPackedDSSpec
                            (pdss: PackedDSSpecPtr): BOOLEAN;
FUNCTION OCEGetDSSpecInfo   (spec: DSSpec): OSType;
FUNCTION OCEGetExtensionType
                            (pdss: PackedDSSpecPtr): OSType;
FUNCTION OCEStreamPackedDSSpec
                            (dss: DSSpec; stream: MyDSSpecStreamer;
                             userData: LONGINT; VAR actualCount: LONGINT):
                             OSErr;
```

Application-Defined Functions

```
FUNCTION MyDSSpecStreamer   (VAR buffer: void; count: LONGINT; eof:
                             BOOLEAN; userData: LONGINT): OSErr;}
```

Assembly Language Summary

Trap Macros Requiring Routine Selectors

`__OCEUtils`

Selector	Routine
$0300	kOCECopyCreationID
$0301	kOCECopyDirDiscriminator
$0302	kOCECopyLocalRecordID
$0303	kOCECopyPackedDSSpec
$0304	kOCECopyPackedPathName
$0305	kOCECopyPackedRLI
$0306	kOCECopyPackedRecordID
$0307	kOCECopyRLI
$0308	kOCECopyRString
$0309	kOCECopyRecordID
$030A	kOCECopyShortRecordID
$030B	kOCEDuplicateRLI
$030C	kOCEEqualCreationID
$030D	kOCEEqualDirDiscriminator
$030E	kOCEEqualDSSpec
$030F	kOCEEqualLocalRecordID
$0310	kOCEEqualPackedDSSpec
$0311	kOCEEqualPackedPathName
$0312	kOCEEqualPackedRecordID
$0313	kOCEEqualPackedRLI
$0314	kOCEEqualRecordID
$0315	kOCEEqualRLI
$0316	kOCEEqualRString
$0317	kOCEEqualShortRecordID
$0318	kOCEExtractAlias
$0319	kOCEGetDSSpecInfo
$031A	kOCEGetIndAttributeType
$031B	kOCEGetIndRecordType
$031C	kOCEGetXtnType
$031D	kOCEIsNullPackedPathName
$031E	kOCENewLocalRecordID
$031F	kOCENewRLI

Selector	Routine
$0320	kOCENewRecordID
$0321	kOCENewShortRecordID
$0322	kOCEPackDSSpec
$0323	kOCEPackPathName
$0324	kOCEPackRLI
$0325	kOCEPackRLIParts
$0326	kOCEPackRecordID
$0327	kOCEPackedDSSpecSize
$0328	kOCEPackedPathNameSize
$0329	kOCEPackedRLIPartsSize
$032A	kOCEPackedRLISize
$032B	kOCEPackedRecordIDSize
$032C	kOCEDNodeNameCount
$032D	kOCERelRString
$032E	kOCESetCreationIDtoNull
$032F	kOCEUnpackDSSpec
$0330	kOCEUnpackPathName
$0331	kOCEUnpackRLI
$0332	kOCEUnpackRecordID
$0333	kOCEValidPackedDSSpec
$0334	kOCEValidPackedPathName
$0335	kOCEValidPackedRecordID
$0336	kOCEValidPackedRLI
$0337	kOCEValidRLI
$0338	kOCEValidRString
$0339	kOCECToRString
$033A	kOCEPToRString
$033B	kOCERToPString
$033C	kOCEPathFinderCID
$033D	kOCEStreamPackedDSSpec
$0344	kOCENullCID
$0345	kOCEGetAccessControlDSSpec
$0346	kOCEGetRootPackedRLI

Result Codes

There is no allocated range of result codes for the Utility Manager. Functions may, however, return standard Macintosh result codes such as `noErr` 0 (No error) and `memFullErr` –108 (Buffer not large enough).